Объектно-ориентированные методы

И. Грэхем

ВИЛЬЯМС

Addison Wesley

Объектно-ориентированные методы

Принципы и практика

Третье издание

ИАН ГРЭХЕМ

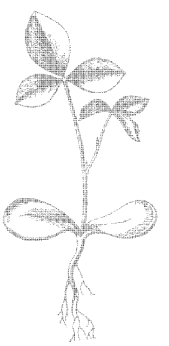
ВИЛЬЯМС

Addison Wesley

Объектно-ориентированные методы

Принципы и практика

Третье издание



Object-Oriented Methods

Principles & Practice

Third Edition

Ian Graham

ADDISON-WESLEY

An imprint of Pearson Education
Harlow, England · London · New York · Reading, Massachusetts · San Francisco
Toronto · Don Mills, Ontario · Sydney · Tokyo · Singapore · Hong Kong · Seoul
Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan

Объектно-ориентированные методы

Принципы и практика

Третье издание

Иан Грэхем

Издательский дом "Вильямс"
Москва ♦ Санкт-Петербург ♦ Киев
2004

ББК 32.973.26-018.2.75
Г91
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *С.В. Беликовой, Н.А. Голобородько, И.Ю. Дорошенко, Р.Г. Имамутдиновой*, докт. техн. наук *Н.Н. Куссуль, М.А. Сидоренко, О.М. Ядренко, В.Д. Яновской*

Под редакцией докт. техн. наук *Н.Н. Куссуль*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Г91 Объектно-ориентированные методы. Принципы и практика. 3-е издание. : Пер. с англ. — М. : Издательский дом "Вильямс", 2004. — 880 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0438-2 (рус.)

Новое издание этой весьма популярной книги было полностью переработано автором с целью отразить все те значительные изменения, которые произошли в объектно-ориентированной методологии с момента выхода предыдущего издания (1991 г.). В частности, большое внимание здесь уделяется многоуровневому проектированию и компонентной технологии, языкам Java и UML. В новом издании обсуждаются все ключевые концепции, преимущества и недостатки, свойственные объектно-ориентированному подходу, а также описываются технологии и инструменты, доступные разработчику в настоящее время.

Книга будет полезна как специалистам-профессионалам, так и тем, кто только приступает к изучению методологии ООП.

ББК 32.973.26-018.2.75

ISBN 5-8459-0438-2 (рус.)
ISBN 0-201-61913-X (англ.)

Оглавление

Содержание

Эта книга посвящается моему сыну, Роберту Грэхему Миллеру.

Предисловие

Предисловие к третьему изданию

Он отбросил книгу, вытянул ноги к угольям, тлеющим в камине, и сплел пальцы за головой, погрузившись в то дивное чувство, которое овладевает человеком, когда мысль от отдельно взятого предмета вдруг совершает скачок к постижению глубинного смысла его связей со всем бытием.

Дж. Элиот. *Миддлмарч*

В 1991 году, когда вышло первое издание этой книги, в среде специалистов по информационным технологиям, а также среди ученых царила атмосфера всеобщей увлеченности объектно-ориентированными методами. Почти каждую неделю проводились какие-либо открытые для широкой публики семинары, посвященные объектно-ориентированному подходу (ООП). Создавались новые журналы, специализирующиеся на данной теме, одна за другой проводились конференции, а число членов всевозможных групп и обществ, связанных с модным тогда направлением развития информационных технологий, стремительно росло. К моменту выхода второго издания интерес к ООП достиг своего апогея, и трудно было бы найти коммерческую организацию, не имевшую хотя бы первого опыта применения данной технологии (то ли положительного, то ли отрицательного). "Объектно-ориентированная лихорадка" продолжается и по сей день. В то время как в области языков программирования достигнута хотя бы относительная стабильность, в области методов проектирования и моделей жизненного цикла программных систем наблюдается настоящий бум. Это происходит даже несмотря на появление универсального языка моделирования UML (Unified Modelling Language), который внес определенную ясность в дискуссии о подходящей системе обозначений. На сегодняшний день акцент сместился на технологии разработки интегрированных приложений корпоративного уровня и компонентные технологии. Со времени выхода второго издания область объектных технологий (ОТ), по видимому, увеличилась втрое, если судить по числу опубликованных работ и освещенных в них вопросов. И вот начала вырисовываться пугающая перспектива основательной доработки этой книги, призванной давать исчерпывающее описание современного состояния ООП, что было бы почти невозможно сделать без существенного увеличения ее объема. С другой стороны, многие из новейших разработок являются, в сущности, вариациями

на темы, известные еще с 1994 года. Так, популярность понятия идиомы, введенного Джимом Коплиеном (Jim Coplien) для языка C++, стала одним из первых предвестников возникновения настоящего интереса к шаблонам проектирования. Ранние версии брокеров объектных запросов, основанные на архитектурной модели группы OMG, постепенно стали широко применяться при разработке программного обеспечения. Кроме того, с учетом накопленного опыта были созданы новые, более совершенные языки объектно-ориентированного программирования, а объектно-ориентированные базы данных "покинули" исследовательские лаборатории и стали использоваться в коммерческих целях, хотя и не так широко, как ожидалось. Вместе с тем исследования в области методов анализа и проектирования отличаются радикальной новизной, и подобных очевидных преемственных связей проследить не удается. В итоге, переработка книги обещала быть весьма существенной.

Предмет книги

Данная книга представляет собой обзор всей области объектных технологий. Здесь рассматриваются вопросы объектно-ориентированного программирования, объектно-ориентированного проектирования, объектно-ориентированного анализа и объектно-ориентированных баз данных. Кроме того, в этой книге затрагиваются и некоторые смежные технологии. Существует достаточно много других книг по объектному подходу, посвященных подробному описанию тех или иных языков или методов. Однако более широкие концепции излагаются в них лишь мимоходом. Как правило, в этих книгах содержится абстрактное высокоуровневое описание "философии" объектно-ориентированного программирования и его преимуществ, однако читателю при этом сразу же предлагается большое количество специфического материала, связанного с синтаксисом определенного языка программирования. С другой стороны, в последнее время появились превосходные и исчерпывающие руководства по вопросам менеджмента в информационных технологиях, однако этим книгам недостает конкретного технического материала, необходимого студентам и программистам-практикам. Читатели, желающие постичь те аспекты объектной технологии, которые напрямую не связаны с технической стороной программирования, вынуждены обращаться к научной литературе, трудам конференций, объемным монографиям или сборникам статей, насыщенных изощренными выкладками. Таким образом, для охвата всей области объектных методов и предвидения их будущей роли необходимо продвигаться несколькими параллельными путями, не забывая при этом об их взаимосвязанности. Поэтому задача данной книги состоит в том, чтобы восполнить имеющийся в литературе пробел. Это достигается следующим образом.

- Книга представляет собой целостный, исчерпывающий, независимый от конкретного языка программирования обзор всех аспектов объектной технологии, причем как с точки зрения менеджера, так и с точки зрения программиста-разработчика.
- Наибольшее внимание уделяется концептуальному моделированию, а не проектированию или собственно программированию. Предпочтение отдается также вопросам, связанным с практическим применением объектно-ориентированных технологий в процессе разработки коммерческих программных систем.
- Авторы книги считают, что ядро современных информационных технологий составляют объектные методы, искусственный интеллект и модели данных, причем рассматриваемые не по отдельности, а в неразрывной связи.

- В книге содержится введение в область объектно-ориентированных языков, инструментальных средств проектирования, баз данных и методов, а также прогнозируются будущие пути их развития и рассматривается их связь с традиционными методами. В частности, вкратце рассказывается о мощном методе Catalysis™ компонентной разработки программных систем [204].
- В книге предпринимается также попытка развеять некоторые мифы относительно объектных технологий, при этом перспектива их практического применения оценивается достаточно положительно.
- Наконец, книга предоставляет читателю достаточно богатый справочный материал и обширную библиографию, что позволит заинтересованному студенту или программисту-практику самостоятельно углублять свои знания.

Еще одна задача данного издания, как и двух предыдущих, состоит в том, чтобы дать четкие ответы на следующие вопросы.

- Какие существуют объектно-ориентированные методы?
- Каковы их преимущества, недостатки и возможные накладные расходы?
- Какие существуют языки, методы и инструментальные средства объектно-ориентированного программирования и по каким критериям их можно оценить?
- Что необходимо сделать, чтобы перейти к использованию того или иного метода или языка программирования?
- Какова роль методов объектно-ориентированного анализа и проектирования?
- Как выявить и выразить требования к разрабатываемой объектно-ориентированной системе?
- Как управлять разработкой объектно-ориентированных систем?
- Какие знания для этого требуются?
- Как объектные методы связаны с другими областями информационных технологий (ИТ)?
- Для разработки каких приложений особенно эффективна объектно-ориентированная технология?

Помимо перечисленных выше задач, данная книга имеет еще одну: ознакомить читателя с моими собственными, основанными на идее использования наборов правил (ruleset), результатами в области объектно-ориентированного концептуального моделирования, проектирования требований и процесса разработки — подход, получивший название SOMA (Semantic Object Modelling Approach — семантический подход к объектному моделированию). Мне удалось соединить методы SOMA, Catalysis и язык UML, который используется при изложении данного материала. При изучении соответствующих глав книги читателю следует иметь в виду одну отличительную особенность моего метода, которая состоит в трактовке объектного моделирования как универсального способа представления знаний о предметной области, а не способа описания структур программ.

Основные отличия от предыдущих изданий

Настоящее издание было существенно переработано и дополнено новым материалом, отражающим основные изменения, которые произошли в области объектных технологий со времени выхода предыдущих изданий. Быстрый и повсеместный переход разработчиков программного обеспечения на объектные технологии изумил не только меня, но и самых больших приверженцев этого нового направления. Значительные изменения в самой технологии, которые произошли в течение последних трех лет (или около того), были уже менее удивительны. Эти изменения заключаются не только в росте числа созданных программных продуктов и используемых при этом методов. По сравнению с 1991 или 1994 годом, ситуация изменилась в целом, и вдумчивый аналитик или программист-практик сможет без труда найти достаточно много тому доказательств. Одним из наиболее значимых явлений стало широкое признание представителями программной индустрии разработок группы OMG, в особенности предложенных ею различных стандартов в области объектных технологий. С другой стороны, многое осталось на своих местах. В общем, задача нового издания книги остается прежней, но для ее достижения требуются совсем другие способы. В настоящем издании пересмотрены многие ключевые понятия, чтобы привести их в соответствие современному уровню и новым стандартам. Кроме того, обновлено описание многих доступных средств и методов разработки, а на основе новых фактов сформулированы новые выводы.

Основные изменения состоят в следующем. Глава 1 подверглась лишь незначительной переработке с тем, чтобы добиться большей ясности и единообразия той терминологии, которая сложилась в области разработки программного обеспечения на момент написания первого издания. Эти уточненные формулировки были выработаны мной благодаря опыту применения более совершенных (как я надеюсь) педагогических методик при формировании курсов лекций по данному предмету и представлении их перед широкой аудиторией. В новой главе 4, посвященной промежуточным звеньям и стратегиям передачи данных, содержится обширный материал по стандартам группы OMG. Материал по объектно-ориентированным базам данных из главы 5 был полностью переработан и теперь охватывает новейшие и гораздо более совершенные разработки в этой области. Наиболее существенной переработке подверглись главы по объектно-ориентированному анализу, проектированию, а также организации процессов и управлению ими. Сделанный в предыдущем издании обзор более чем 50 методов, актуальных на то время, теперь представляет лишь исторический интерес и поэтому помещен в приложение. Теперь во всей книге используется получивший широкое распространение язык UML, а в главах 6 и 7 рассматриваются реальные примеры хорошо выполненного объектно-ориентированного анализа и проектирования, в основу которых положены принципы методов Catalysis и SOMA. В отдельном приложении содержится краткое описание языка UML. В новой главе 7 рассматриваются архитектура программных систем, шаблоны проектирования и принципы разработки на основе компонентов. В главе 8 подробно описан подход SOMA к разработке на основе требований. Глава 9, посвященная менеджменту, в значительной степени переработана для достижения большей ясности в изложении материала. В частности, в ней содержатся более конкретные практические рекомендации по организации процесса разработки программной системы. В данную главу включены также рекомендации по проектированию пользовательского интерфейса. Во все остальные главы и приложение А внесены относительно небольшие изменения и дополнения, касающиеся новых разработок в данной области. Кроме того, были исправлены известные мне ошибки, обнаруженные во втором издании.

На протяжении последних лет все большее внимание я уделяю той многочисленной категории читателей, которые используют мои книги в качестве учебных пособий. Специально для них в конце каждой главы я добавил упражнения. Ответы на некоторые из них (по крайней мере, если ответ известен мне) можно найти на Web-узле компании TriReme. Объем библиографии заметно возрос по сравнению с предыдущим изданием, что вполне соответствует потоку новой литературы и расширению самой рассматриваемой области. Новый вариант словаря терминов также заметно отличается от предыдущего.

Несмотря на столь глубокую переработку, тема и основные задачи книги остаются неизменными, и я надеюсь, что в этом издании мне удалось сделать исчерпывающий, еще более ясный, актуальный и точный обзор области объектно-ориентированных методов.

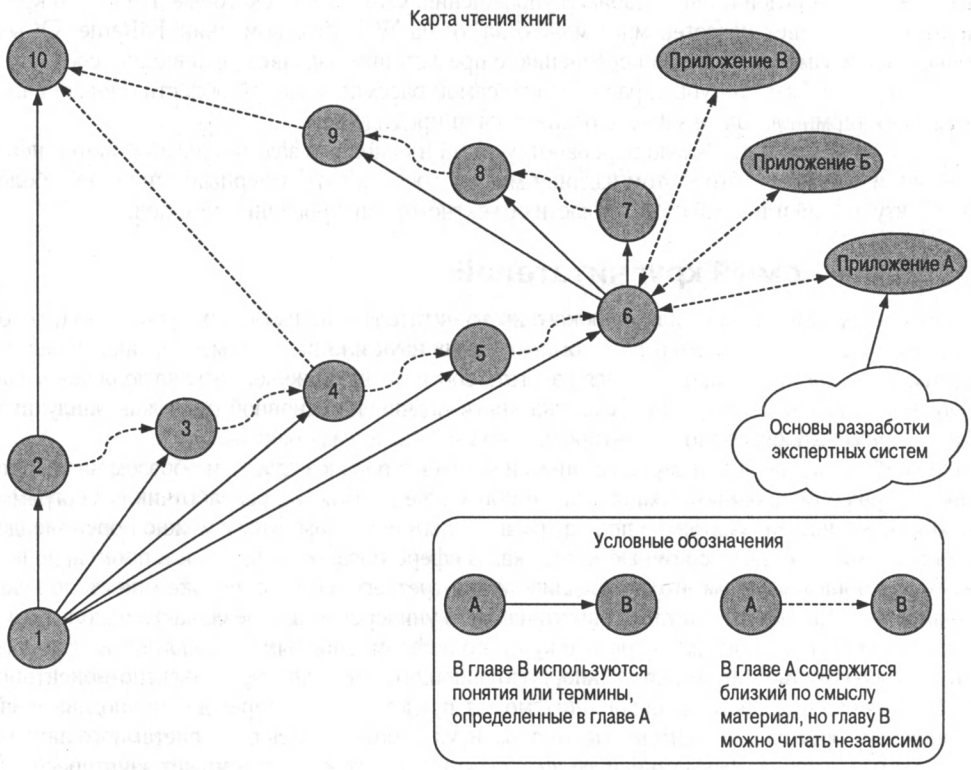
Предполагаемый круг читателей

Эта книга предназначена для широкого круга читателей, не имеющих профессиональной подготовки в области теоретических компьютерных наук или высшей математики. Вместе с тем, там, где это требуется, я старался излагать соответствующие вопросы на должном уровне глубины и научной строгости. Тематика книги подчинена основной цели: как наилучшим образом применять объектно-ориентированную технологию на практике.

При работе над первыми двумя изданиями я ориентировался главным образом на специалистов по информационным технологиям и обработке данных, на разработчиков программного обеспечения, консультантов по информационным системам. Книгу можно порекомендовать всем, кто имеет дело с компьютерами как в сфере образования, так и в промышленности. Хотя в общем и целом это справедливо и для третьего издания, все же мне стало ясно, что наиболее благодарного читателя книга нашла в университетах, где ее часто используют в качестве учебного пособия для вводного курса по информационным технологиям или по разработке программных систем, возможно, дополняющего еще один курс объектно-ориентированного программирования. Данная книга может представлять интерес для преподавателей, специализирующихся в области компьютерных наук (Computer Science), системного анализа в бизнесе и, возможно, искусственного интеллекта. Ученых эта книга может заинтересовать как сжатый обзор-справочник современных результатов с указанием авторов оригинальных работ. Возможно, их также заинтересуют рассыпанные по тексту книги комментарии, в которых содержатся нестандартные оценки существующих работ, развивающие творческое начало. Прочитав эту книгу, менеджеры и руководители проектов смогут лучше понять, как применяемая технология разработки отражается на бизнесе, а значит, смогут более эффективно планировать свою деятельность. Консультантам, менеджерам проектов, системным аналитикам и проектировщикам книга поможет составить более точное и полное представление о современных технологиях и, таким образом, идти в ногу со временем, на практике применяя описанные в книге методики. Программисты, прочитав эту книгу, смогут расширить свой кругозор.

По мере создания материал книги проходил "обкатку" перед широкой аудиторией на различных конференциях, семинарах и учебных курсах.

Рекомендуемый порядок чтения



Книга построена таким образом, чтобы материал можно было читать в различном порядке. Два основных типа зависимостей между главами, влияющие на порядок чтения, представлены на диаграмме. Менеджерам и всем читателям, которых, в первую очередь, интересует общая характеристика отрасли на верхнем уровне абстракции, можно порекомендовать двигаться по вертикальной стрелке (главы 1–2 и 10). Менеджерам проектов может пригодиться также глава 9. Читателям, интересующимся методами анализа, следует избрать нижние стрелки, поскольку в главах 6–8 используется материал всех предыдущих глав. Главу 1 настоятельно рекомендуется прочесть всем, даже тем, кто хорошо знаком с объектно-ориентированным программированием, поскольку в ней вводится специфическая терминология, которая может отличаться от используемой в других изданиях; как уже отмечалось, данная книга написана с позиций концептуального моделирования. Темы, представляющие интерес для сравнительно узких специалистов, а также отступления от основной темы помечены специальной пиктограммой.

Я надеюсь, что при последовательном чтении читатель воспримет эту книгу как целостное и связное повествование. Как и всякое повествование, книга имеет несколько основных "сюжетных линий", к которым относятся установка различий в подходах проектировщика и

автора концептуальной модели системы, необходимость интеграции объектно-ориентированного подхода с плодотворными идеями, выработанными в других областях компьютерных наук, а также необходимость добавления к известным преимуществам объектного подхода (таким, как возможность повторного использования результатов и расширяемость построенных систем). Кроме того, данная книга имеет высокую семантическую выразительность.

Замечания относительно орфографии и терминологии

Поскольку многие из нас являются ревностными поборниками строгих правил использования языка и поскольку особенности моего изложения материала уже вызвали нарекания со стороны читателей, я счел необходимым дать разъяснение тем принципам, которыми я руководствовался при написании данного текста.

В данном издании заимствованные слова я пишу в соответствии с греческим или латинским словом-оригиналом, даже если это противоречит привычным правилам. При этом заимствованные слова всюду выделены *курсивом*, их толкование всегда можно найти в словарях иностранных слов. Курсивом также выделены термины, которые являются предметом рассмотрения текущего раздела, и термины определений.

В тексте слово *данные* является формой множественного числа от *данное*, и поэтому я обращался с ним соответственным образом. Мне трудно понять, почему большинство современных авторов, специалистов по компьютерным наукам, допускают одну и ту же ошибку, обращаясь со словом *данные* так, как если бы оно было формой единственного числа, в то время как специалисты в других областях этой ошибки не делают. Я всюду старательно избегал слов с неоднозначным смыслом.

Слово *метод*, одно из наиболее часто встречающихся в книге, имеет различный смысл:

- общее понятие метода разработки программных систем (именно это подразумевает название книги);
- тот или иной конкретный метод разработки программ, например метод Catalysis или SOMA;
- и наконец, программная единица, отвечающая за реализацию поведения объекта.

Я всячески старался избегать употребления некорректного слова *методологии* (во множественном числе) — это такая же нелепость, как и говорить во множественном числе о *физиках* или *химиях*. Напомню, что методология — это общее понятие, означающее *науку о методе* какой-либо науки или научной дисциплины. Единственный случай, когда можно допустить множественное число, связан с так называемым *парадигмным сдвигом*[1] (в смысле Томаса Куна — Thomas Kuhn) в нашем понимании предмета. Так, поскольку большинство людей считают объектную технологию именно таким парадигмным сдвигом, корректной будет фраза вроде *имеется различие между структурной и объектно-ориентированной методологиями*, где речь идет о глобальной смене направлений в компьютерной науке в целом. Однако совершенно неприемлема грамматически сходная фраза *имеется различие между методологиями Йордона и Джексона*. Кстати, слова *парадигма*, получившего столь широкое распространение в компьютерных науках, я также старался избегать.

---

[1] В самом названии можно усмотреть смысловую ошибку. Напомним, что парадигма в общем случае есть модель, образец или типичный пример чего-либо.

При написании книги, как и в жизни, я старался избавиться от укоренившегося в английском языке наследия женоненавистничества. Для обозначения некоторого неопределенного лица я предпочитаю использовать местоимение *он(а)* вместо неудобоваримого оборота *он или она*. Впрочем, и здесь возникают определенные сложности, связанные с различием грамматического рода и биологического пола. Существительное мужского рода *человек* я употребляю для обозначения представителей рода человеческого обоих полов. То же относится и к существительному женского рода *персона* (если последовательно и неуклонно стремиться к совпадению грамматического рода с биологическим полом, французам следовало бы говорить "le ou la personne", а англичанам — ввести в язык новое слово *"woperson"*). Впрочем, истинная политическая корректность обеспечивается изгнанием зла из своих поступков, а не перекраиванием языка.

При изложении текста я вовсе не следовал принципу использования коротких и простых предложений, что характерно для журналистского стиля, за исключением случаев, когда это оправдано большей ясностью. При выборе терминологии я стремился использовать наиболее *правильное* слово, а не наиболее известное. Специфичные и малораспространенные термины, конкретный смысл которых в данном контексте было бы трудно установить с помощью толковых словарей, помещены в глоссарий или разъяснены в подстрочных примечаниях. Вообще говоря, при написании подобной узкоспециализированной книги практически невозможно обойтись без таких терминов, смысл которых в данной области отличается от данного в словарях общей лексики. Так, словосочетание *рабочая станция* имеет в компьютерных науках особый смысл, отличный от смысла таких обыденных слов, как *работа* и *станция*. Впрочем, язык — постоянно развивающийся живой организм, и мы не только пользуемся им, но и создаем его.

Все сказанное выше есть теоретическая база, на которой построено данное изложение. На практике, возможно, текст содержит некоторые грамматические и орфографические ошибки.

Благодарности

Хотя в книге представлен обширный оригинальный материал, принадлежащий автору, все же данное издание было бы невозможно без работ других авторов, которым я выражаю свою искреннюю признательность. Кроме того, многие из высказанных в книге идей возникли в процессе личного общения автора с коллегами и друзьями.

Я особо благодарен Акмалю Шаудри (Akmal Chaudri) за его вклад в написание главы 5. В работе над этой главой пригодились замечания, авторы которых мне, к сожалению, неизвестны. Глава 7 написана совместно с Аланом О'Каллаганом (Alan O'Callaghan) из университета Де Монфорт (De Montfort). Алан Камерон Уиллс (Alan Cameron Wills) выступил соавтором глав 6 и 7. Его участие заслуживает особого упоминания: большая часть иллюстраций в этих двух главах создана им для учебных курсов компании TriReme по языку UML и методу Catalysis. Алан до сих пор продолжает обучать меня данному предмету, и я признателен ему за полезную информацию и за моральную поддержку при написании этой книги. Впрочем, сказанное не означает, что мои соавторы согласны со всеми моими утверждениями, представленными в книге, или полностью разделяют мою позицию. Ответственность за все ошибки, которые остались в настоящем издании, всецело лежит на мне.

Третье издание имеет столь долгую историю и предысторию, что невозможно перечислить всех, кому я благодарен за вклад в его создание. Однако я не могу не выразить свою благодарность следующим людям: Марк Аддисон (Mark Addison), Нигел Бэкхарст (Nigel

Backhurst), Сэв Боджа (Sav Bhoja), Стив Биркбек (Steve Birkbeck), Джулия Бишоф (Julia Bischof), Гради Буч (Grady Booch), Франко Чивелло (Franco Civello), Дэйв Кларк (Dave Clark), Алистер Кокбурн (Alistair Cockburn), Грэм Коллинз (Graham Collins), Ларри Константайн (Larry Constantine), Кос Константину (Cos Constantinou), Джон Крессвелл (John Cresswell), Робин Круи (Robin Crewe), Джон Дэниелс (John Daniels), Сэлли Дэвис (Sally Davies), Малькомб Дик (Malcomb Dick), Фарамарз Фаруди (Faramarz Farhoodi), Армандо Феррейра (Armando Ferreira), Дон Файерсмит (Don Firesmith), Мартин Фоулер (Martin Fowler), Флоранс Фруадево (Florence Froidevaux), Стюарт Фрост (Stuart Frost), Тим Ги (Tim Gee), Томас Гротейн (Thomas Grotehen), Брайан Хендерсон-Селлерс (Brian Henderson-Sellers), Кевлин Хенни (Kevlin Henney), Крис Харрис-Джонс (Chris Harris-Jones), Бенедикт Хил (Benedict Heal), Тим Лэмб (Tim Lamb), Дэвид Харви (David Harvey), Джейн Хиллстон (Jane Hillston), Майкл Джексон (Michael Jackson), Маргарет Джеймс (Margaret James), Питер Джонс (Peter Jones), Фиона Киннер (Fiona Kinnear), Дэвид Ли (David Lee), Крис Лис (Chris Lees), Марк Льюис (Mark Lewis), Ник Люкик (Nick Lukic), Маргарет Лиолл (Margaret Lyall), Нейл Мейден (Neil Maiden), Соня Мат (Sonia Math), Клив Менхиник (Clive Menhinick), Сэлли Мортимор (Sally Mortimore), Дерек Пирс (Derek Pearce), Колин Проссер (Colin Prosser), Роб Радмор (Rob Radmore), Аль-Нур Рамджи (Al-Noor Ramji), Дэн Роусторн (Dan Rowsthorne), Дэвид Рэдмонд-Пил (David Redmond-Pyle), Элейн Ричардсон (Elaine Richardson), Рене Шварб (René Schwarb), Ричард Сид (Richard Seed), Тони Симонс (Tony Simons), Ричард Смит (Richard Smith), Гейл Сваффилд (Gail Swaffield), Джон Тейлор (John Taylor), Брайан Тал (Brian Thal), Ульрика Тиссен (Ulrika Thyssen), Ричард Уолкер (Richard Walker), Роуз Ватсон (Rose Watson), Джон Велч (John Welch), Дэвид Велтон (David Welton), Керри Вильямс (Kerry Williams), Бенуа Ксенсеваль (Benoit Xhenseval), Дженифер Йейтс (Jennifer Yates), Маркус Земп. Стоит упомянуть также следующие организации: BIS Applied Systems, BIS Banking Systems, Chase Manhattan Bank, De Montfort University, Eric Leach Marketing, Equitable Life, Swiss Bank Corporation (ныне UBS), TriReme International, а также секцию экспертных систем, объектно-ориентированного программирования и проектирования на основе требований Британского компьютерного общества. Мне было приятно работать с сотрудниками издательства Pearson Education, и особую благодарность хотелось бы выразить редактору Элисон Биртвелл (Alison Birtwell).

И наконец, пришло время признаться в тех теплых чувствах, которые я питаю к кабачку "The Grove" ("Роща"), где я создал большую часть моих книг. Это викторианский паб со старыми добрыми традициями и очаровательным интерьером, и в нем почти всегда достаточно пусто, чтобы я мог рассчитывать на свободный столик для работы. Возможно, именно по этой причине некоторые завсегдатаи, представители самых разных профессий, от автомехаников до криминалистов, называют этот паб "The Grave" ("Могила"). Впрочем, другие авторы, которые приходят сюда работать, сравнивают его с читальным залом Бэлхэмской библиотеки. Насколько я знаю, в этом уютном кабачке написаны по меньшей мере две монографии по культуре Соломоновых островов и один роман. Там родилась также одна песня "Баллада о Кене Ливингстоне" (текст могу выслать по заявке). Кроме того, посещение паба дает мне редкую возможность присоединиться к облюбовавшему это место ансамблю "Wandle Delta String Band" и вдоволь поиграть на бодхране — а заодно и на нервах публики. До встречи в "Роще"!

Иан Грэхем (Ian Graham)
Бэлхэм, ноябрь 2000 г.
(ian@trireme.com)

1

Основные понятия

Мои объекты вечны,
Их нужно сохранить.

В. С. Гилберт. *Микадо*

В 1990-х годах словосочетание "объектно-ориентированный" в контексте информационных технологий стало синонимом слов "современность", "высокое качество", "ценность". Некоторое время спустя слово "объект" стало заменяться словом "компонент", хотя и в несколько ином значении. Поскольку мы склонны к преувеличению достоинств этой технологии, желательно было бы добиться более объективной точки зрения. Эта книга призвана сформулировать принципы объектно-ориентированной и модульной разработок в корректных терминах. Кроме того, она должна показать, как эти абстрактные идеи могут превратиться в реальные, практические советы по построению полезных компьютерных систем. Описываемый подход не зависит от конкретного языка программирования и не охватывает вопросы синтаксических предпочтений и выбора среды разработки. При использовании объектно-ориентированных сущностей нужно также понимать, что мы вступаем в область, уже имеющую некоторые стандарты. Однако в настоящее время ее границы еще не определены, а исследования остаются незавершенными. Непременно нужно рассмотреть множество других сфер, связанных с данной, таких как архитектура программного обеспечения, распределенные открытые системы, системы баз данных, CASE-технологии, экспертные системы и т.д. Придется также столкнуться и со сравнительно неизвестными сферами.

Ключевые преимущества объектно-ориентированного подхода — это возможность повторного использования и расширяемость, т.е. объектно-ориентированные системы могут быть легко собраны из ранее написанных компонентов. Эти системы будут легко расширяться без какой-либо модернизации повторно используемых компонентов. В данной главе рассмотрены оба указанных преимущества, а также область применения объектно-ориентированных систем.

Объектно-ориентированным методом в этой книге называется нечто большее, чем просто объектно-ориентированное программирование. В это понятие включается философия разработки систем программирования, извлечение знаний, анализ требований, моделирование предметной области, проектирование систем, проектирование баз данных и многие другие вопросы. Основное внимание будет обращаться на указанную философию и ее использование для решения проблем, возникающих при создании информационных систем. Например, такие преимущества, как возможность повторного использования и расширяемость, не ограничиваются повторным использованием или дополнением частей кода. Потенциально документы по проектированию и анализу могут храниться в библиотеках и повторно использоваться или расширяться снова и снова, если, разумеется, пользователи смогут их легко отыскать.

Как уже отмечалось, в термин "объектно-ориентированный" вкладывают слишком общий смысл. Чтобы лучше понять данный вопрос, в этой книге будут различаться понятия "объектно-ориентированное" (object-oriented), "компонентно-ориентированное" (component-oriented), "основанное на объектах" (object-based) и "основанное на классах" (class-based) программирование, а также их разработка и анализ. В следующих главах будет показано, что некоторые коммерческие системы действуют согласно объектно-ориентированным принципам, но имеют некоторые недостатки. Тем не менее объектно-ориентированные системы на самом деле существуют, и важно не то, является ли система или язык объектно-ориентированным, а то, *как* они реализованы.

В этой главе вводятся основные понятия и терминология объектно-ориентированных методов. Вначале приводится краткая историческая справка. Мотивация и преимущества объектно-ориентированных методов и программирования будут изложены в главе 2.

1.1. Историческая справка

История развития объектно-ориентированного подхода отражает и повторяет историю вычислительной техники в целом. Начнем с 1940-х годов, когда первые работы по вычислительной технике были связаны исключительно с тем, что в настоящее время называют программированием. Только позже выделились проектирование и анализ. Точно так же первым привлекло к себе внимание объектно-ориентированное программирование, позже появилось объектно-ориентированное проектирование, а еще позже объектно-ориентированный анализ. Таким образом, эта книга начинается с описания объектно-ориентированного программирования, затем рассматриваются вопросы разработки и анализа, хотя далее основное внимание будет уделено именно последнему.

Хотя Тен Дайк (Ten Dyke) и Канц (Kunz) объявили, что разработчики ракет Minuteman использовали элементарные объектно-ориентированные методы еще в 1957 году, история объектно-ориентированного программирования на самом деле началась в Норвегии в 1967 году. Однако с развития языка программирования Simula, основанного на языке ALGOL и более раннем языке моделирования дискретных событий Simula 1, и продолжался использоваться в 1970-х годах объектный подход параллельно с языком Smalltalk, который сделал понятие "объект" объектом поклонения. Важными промежуточными этапами были языки Alphard [810] и CLU [490]. Стоит отметить, что в объектно-ориентированном языке Simula были представлены все понятия структурного программирования. С тех пор было создано много языков, которые были порождены этими разработками и получили название "объектно-ориентированных".

Имитационное моделирование — трудная задача для программистов, использующих обычные языки третьего поколения. От программистов требуется адаптировать функциональный поток управления, обычный для таких языков, к потоку управления, который естественнее описывается через составные объекты, изменяющие свое состояние, и происходящие события. В объектно-ориентированном программировании функциональный поток заменяется передачей сообщений между объектами, которые вызывают изменения состояния. Таким образом, объектно-ориентированное программирование — это крайне естественный подход, поскольку структура программ непосредственно отражает структуру задачи. Более того, в моделируемых задачах обычно понятно, что является объектами. В частности, это могут быть машины на улице, механизмы производственной линии и т.д. Они обычно являются "реальными", а не абстрактными объектами, и как таковые, их легче определить. К сожалению, как будет показано далее, это не всегда справедливо для коммерческих приложений.

Smalltalk и GUI

Термин "объектно-ориентированный" окончательно утвердился с появлением языка программирования Smalltalk, который был создан в исследовательском центре Xerox в Пало Альмо. Он основывался не только на языке Simula, но и на диссертации Алана Кая (Alan Kay), выполненной в университете штата Юта. Как отмечено в работе [659], эти исследования были основаны на представлении маленького, но универсального персонального компьютера, способного решать любые задачи управления информацией. Первой версией подобного устройства была машина Flex, которая получила название Dynabook. Smalltalk — это, по сути, программное обеспечение для Dynabook, сильно зависимое от понятия классов языка Simula, а также понятия наследования и структурных особенностей языка Lisp[1]. Smalltalk объединил понятие класса из Simula с набором функциональных абстракций, подобных Lisp, хотя, как языки программирования, Smalltalk и Lisp достаточно не похожи.

Следующий этап, который пришелся на 1980-е годы, продемонстрировал рост интереса к интерфейсу пользователя (UI). Самый известный пионер в области коммерческих продуктов, компания Xerox, а позже и компания Apple заинтересовали мир удобным WIMP[2]-интерфейсом, и много идей в Smalltalk прочно связано с этими разработками. С одной стороны, объектно-ориентированное программирование способствовало разработке таких интерфейсов — особенно это касается Lisa и Macintosh от компании Apple — а с другой, стиль объектно-ориентированных языков находился под влиянием идеологии WIMP. Наиболее очевидное следствие этого — это большое (по сравнению с другими сферами) множество библиотечных объектов для разработки интерфейса. Одной из основных причин успеха объектно-ориентированного программирования была сложность таких интерфейсов и сопутствующая этому высокая цена реализации. Можно предположить, что без присущей объектно-ориентированному коду возможности повторного использования было бы невозможно обеспечить столь широкое распространение таких интерфейсов. Например, известно, что для разработки Apple Lisa, предшественника Macintosh, потребовалось более 200 человеко-лет

---

[1] Lisp (LISt Processing) означает "обработка списков". Этот язык, изначально разработанный Джоном Маккарти приблизительно в 1958 году, стал основным языком реализации для многих ранних работ в области искусственного интеллекта.

[2] WIMP — это сокращение от Windows, Icons, Menus (иногда Mice), Pointers, которое обозначает стиль графического интерфейса пользователя (Graphical User Interface — GUI).

работы, большая часть которой ушла на разработку интерфейса. На протяжении этого периода влияние WIMP-стиля было настолько сильным, что все терминалы были постепенно оборудованы внешними интерфейсами WIMP, такими как Microsoft Windows, или им подобными. С учетом курса на стандартизацию в открытых системах на основе UNIX также учитывались аспекты UI, причем в качестве немаловажного фактора выступала конкурентная борьба OpenLook, OSF Motif и других подобных программ. В этом смысле объектно-ориентированный подход наложил свой отпечаток на отрасль компьютеризации и определил тот внешний вид экрана, какой мы привыкли видеть с 1993 года.

Влияние искусственного интеллекта

Начиная с середины 1970-х годов наблюдается значительная взаимосвязь между объектно-ориентированным программированием и исследованием и разработкой систем искусственного интеллекта (ИИ), что привело к появлению нескольких важных языков искусственного интеллекта, в частности Lisp. Именно благодаря этому появился Lisp со средствами Flavors, Loops и CLOS (Common Lisp Object System). Идеи объектно-ориентированного подхода оказали свое влияние на среду программирования ИИ (в том числе расширения Lisp, такие как KEE и ART). На эти системы также сильно повлияли методы представления информации, основанные на семантических сетях и фреймах. Эти структуры выражают знания о реальных объектах и понятиях в виде сетей стандартных объектов, которые могут наследовать свойства от своих родителей. Таким образом, основным вкладом этого расширения объектно-ориентированного подхода стала строгая теория наследования. Объектно-ориентированные методы до сих пор основываются на этих результатах. Языки искусственного интеллекта будут обсуждаться позже, в главе 3.

Еще одно направление исследований в области ИИ, наряду с работами в сфере параллельных вычислений, привело к появлению понятия *исполнитель* (actor). Системы с использованием исполнителей [6], в том числе системы доски объявлений [261], — это попытка смоделировать работу групп сотрудников или экспертов. Исполнитель — это более антропоморфное понятие, чем объект, для которого определены обязанности, потребности и знания о взаимодействии с другими исполнителями. Языки, использующие понятие исполнителя, обычно предназначены для параллельных приложений реального времени. Родственное современное понятие — это интеллектуальные исполнители (или агенты). Понятно, что современные компонентно-ориентированные средства разработки, такие как технология COM+, требуют знания компонентов о возможных участниках взаимодействия. Более подробно об этом будет сказано в главах 6 и 7.

Новые языки

По сравнению с коммерческими приложениями, разработка пользовательских интерфейсов не порождает серьезных проблем, связанных с управлением данными. Вследствие того, что основное внимание обращалось на моделирование проблем и проектирование пользовательского интерфейса, а не на, скажем, управление базами данных, характерным свойством ранних языков программирования была проблема производительности. Это привело к разработке новых языков, таких как Eiffel, и расширению имеющихся, удобных и эффективных языков, таких как Ada, C и PASCAL. Такое смещение внимания также означало, что языки объектно-ориентированного программирования часто не предоставляли средств работы с постоянными объектами, параллелизмом и т.д. Одним из средств решения данной проблемы

стала разработка систем объектно-ориентированных баз данных. Детально этот вопрос будет обсуждаться в главе 5. Некоторые объектно-ориентированные и объектные языки программирования обзорно рассмотрены в главе 3.

Требования многих пользователей, а особенно таких финансово влиятельных, как министерство обороны (DoD) Соединенных Штатов Америки, часто приводят к тому, что промышленные компании вынуждены менять курс. Требования DoD на протяжении 1960–1970-х годов затрагивали три основных аспекта: инженерный подход к практической разработке программного обеспечения, который выразился в так называемых "структурных" методах; возможность повторного использования компонентов программного обеспечения (модульность); и открытость систем. Разнообразные методы разработки систем, принятые многими основными поставщиками информационных технологий, — это реакция на первое требование. В настоящее время (после бума, произведенного CASE-средствами в конце 1980-х) в этой области наблюдается некоторое затишье. Системы UNIX, X/Windows и Ada — все это в некотором роде реакция на требование DoD относительно "открытости" систем. Это требование означает минимизацию затрат на взаимодействие с любым программным обеспечением или аппаратными средствами. Язык Ada также характеризуется улучшенной модульностью. Позже мы оценим, насколько язык Ada является объектно-ориентированным. А сейчас нас интересует, вносит ли он что-то (и объектно-ориентированное программирование вообще) в перечисленные три ключевых вопроса. Объектно-ориентированное программирование также неявно входит в перечисленные выше ключевые требования DoD. Оно дает возможность строить системы с использованием повторно используемых компонентов, объединяя таким образом в единое целое модульность и программную инженерию. С появлением разного рода промежуточного программного обеспечения, ориентированного на сообщения, и стандартов передачи сообщений, таких как CORBA (см. главу 4), возможность взаимодействия в рамках действительно распределенных систем расширяется еще больше.

В процессе становления объектно-ориентированного программирования интерес сместился к объектно-ориентированным методам проектирования и анализа (или описания). Преимущества повторного использования и расширяемости можно рассматривать в ракурсе проектирования и описания, а не только программирования. Авторы работ [76], [643] и [723] соглашаются, что в общем контексте программной инженерии, чем выше уровень повторного использования, тем лучше. Это приводит к возникновению важных вопросов. Должно ли объектно-ориентированное проектирование реализовываться в объектно-ориентированном языке? Должны ли методы проектирования быть связаны с определенными языками?

Новые базы данных и CASE-средства

В начале 1990-х годов, когда реляционные базы данных стали общепринятой (если не необходимой) технологией реализации коммерческих продуктов, их основные производители стали разрабатывать различные "постреляционные расширения" своих продуктов, предназначенные для таких отраслей, как экспертные системы, функциональное программирование, а позже и объектно-ориентированное программирование. Появились коммерческие продукты объектно-ориентированных и полуобъектно-ориентированных баз данных, а теоретические вопросы объектно-ориентированного подхода сместились в сторону традиционных вопросов баз данных. Например, стали актуальными задачи эффективного управления перманентными объектами, кэширования и контроля версий объектов. Эти разработки можно интерпретировать как процесс упорядочения объектно-ориентированного подхода. Возникло также

множество вопросов, касающихся относительной эффективности декларативных языков реляционных запросов по сравнению с подходами, основанными на объектной идентичности. По иронии судьбы последнее свойство относится не только к объектно-ориентированным базам данных, но и к ранним сетевым и иерархическим системам. Недавно стало ясно, что существует фундаментальное различие между двумя подходами: чисто объектно-ориентированными базами данных и смешанными объектно-реляционными системами. Фактически это различие породило два стандарта языков запросов: OQL и SQL3. В главе 5 будет подробнее рассмотрена теория баз данных, а также будут изучены объектно-ориентированные базы данных и другие вопросы.

CASE-средства (computer-aided software engineering) автоматизированного проектирования и создания программ становятся все больше необходимы в разработке коммерческих систем. Одни на это смотрят с восторгом, другие — скептически. Появление многочисленных методов объектно-ориентированного анализа и проектирования и инструментальных CASE-средств, поддерживающих их, заставляет задуматься о преимуществах их использования. В главах 6–8 рассмотрены вопросы объектно-ориентированного анализа и методов проектирования. CASE-средства, поддерживающие эти методы, также описаны в главах 6 и 7.

Распределенные системы и Web

1990-е годы характеризуются возросшими насущными потребностями рынка в разработке нового программного обеспечения, а также появлением более дешевых и более мощных компьютеров. Это привело к новому витку развития объектно-ориентированного подхода и появлению ряда новых его применений, помимо разработки GUI и систем ИИ. Приобрели важное значение распределенные вычисления и архитектура "клиент/сервер", а объектная технология стала основой многих разработок, особенно с появлением так называемых трехуровневых систем на платформе "клиент/сервер". Продолжают играть важную роль и реляционные базы данных. Новые области применения и улучшенные аппаратные средства способствовали массовому использованию объектно-ориентированного программирования и потребовали соответствующего внимания к вопросам объектно-ориентированного проектирования и (позже) анализа. Теория объектно-ориентированных баз данных также была сформирована на протяжении этого десятилетия. Сейчас эти технологии начинают использоваться в широких масштабах. Появление и популяризация World Wide Web привели к возникновению новой проблемы. Поскольку в Web передается самая разнообразная информация — текст, графика, звук или видео — реляционные базы данных не обеспечивают качества, необходимого для приложений, требующих хранения и извлечения сложных структур данных. Для управления этими структурами естественно использовать объектно-ориентированный подход. Первым широко известным языком с поддержкой технологии Web был объектно-ориентированный язык Java. Компании, поддерживающие активно используемые Web-узлы, такие как Microsoft и IBM, для обеспечения репликации, контроля версий, скорости и способности к восстановлению были вынуждены применять объектно-ориентированные базы данных, такие как Versant и ObjectStore. Сетевые вычисления, "тонкие" клиенты и агенты требуют развития общего теоретического подхода и методов разработки программного обеспечения. Перспективы объектной технологии самые обнадеживающие. Развитие данного решения будет представлено далее в этой книге.

Анализ и проектирование

С начала 1990-х основное внимание сместилось с проектирования на анализ. Первая книга под названием *Object-Oriented Systems Analysis* была написана Шлеер и Меллором [707] в 1988 году. Как и в ранней работе Буча (Booch), в ней непосредственно не был описан объектно-ориентированный метод. Основное внимание уделялось описанию расширенной модели "сущность-связь", основанной на представлении задачи в терминах сущностей и отношений между ними. При этом поведенческие аспекты объектов игнорировались. В 1992 году Шлеер и Меллор опубликовали вторую книгу, посвященную своим исследованиям. В этой книге утверждалось, что данное поведение может моделироваться с использованием обычных диаграмм переходов из состояния в состояние. Между тем Питер Коад (Peter Coad) объединил идеи поведения в рамках простого, но объектно-ориентированного метода [171, 172]. Это породило живой интерес к объектно-ориентированному анализу и проектированию и привело к появлению публикаций по этим вопросам.

В последнее время акцент сместился в сторону стандартов. Объектная технология может преуспеть в конкуренции с существующими подходами, только если пользователи получат уверенность в том, что они смогут получить от объектно-ориентированных приложений все необходимые свойства открытых систем. Если объектно-ориентированные приложения взаимно несовместимы, если объектно-ориентированные базы данных не могут взаимодействовать между собой и с реляционными базами данных и если не существует стандартных форм записи и терминологии для объектно-ориентированного анализа, на это надежды мало. Главный сторонник стандартизации — это рабочая группа по развитию стандартов объектного программирования OMG (Object Management Group). OMG — это очень большая группа влиятельных компаний (около 700), призванная добиться согласованности терминологии объектно-ориентированного подхода и стандартов интерфейсов различных поставщиков. Такой уровень сотрудничества — редкость для компьютерной индустрии. Встречи технического комитета OMG проводятся в Европе, США и на Дальнем Востоке на международной основе. Группа OMG призвана быстро публиковать стандарты, быстрее чем какие-либо другие организации по стандартизации. Она уже опубликовала ряд стандартов, начиная с нескольких версий широко используемой технологии построения распределенных объектных приложений CORBA (Common Object Request Broker Architecture) и многоуровневой архитектуры для взаимодействия распределенных объектно-ориентированных приложений, аналогичной, в некотором смысле, семиуровневой модели ISO, и заканчивая стандартизованным определением понятия обмена. Первая версия CORBA определяла принципы построения продуктов, скрывающих сложность стратегии распределения на базе RPC. Стандарт CORBA 2 допускает существование брокеров объектных запросов от разных производителей и их взаимодействие, а в CORBA 3 добавлены языки сценариев и поддержка асинхронной передачи сообщений для их гарантированной доставки. Появилась компонентная модель CORBA Component Model, подобная EJB (Enterprise Java Beans). Это серверные компоненты, которые рассматриваются ниже, в главе 7. Некоторые поставщики предлагают продукты ORB (Object Request Broker), совместимые со стандартами CORBA. Официально ISO признано только несколько стандартов OMG, но все рабочие версии стандартов группы получили широкое распространение. Конкурирующий набор популярных стандартов *де-факто* был разработан в "лагере" Microsoft; к ним относятся компоненты Active X COM+ и DCOM. Эти архитектурные модели, CORBA и DCOM, будут обсуждаться позже, в главе 4. Группа OMG также одобрила язык UML как стандартную форму обозначений для

объектно-ориентированного проектирования, основанную на объединении систем обозначений Буча, Якобсона и OMT. Эти вопросы будут рассматриваться детальнее в главе 6 и приложении Б.

Компоненты

На момент написания книги основными вопросами, связанными с объектной технологией, были компонентная разработка CBD (component-based development), шаблоны, стандартизация формы записи для объектно-ориентированного анализа и проектирования, процесс разработки и архитектура. Эти вопросы рассматриваются в главах 6 и 9. Поскольку объектная технология стала широко распространенной, для организаций, работающих в рамках данного подхода (например, в области электронной коммерции, Web-технологий и промежуточного программного обеспечения), на первый план вышли вопросы совместимости и миграции. Эти вопросы рассматриваются в главах 4 и 5.

Таким образом, самая поздняя фаза развития объектно-ориентированных методов характеризуется смещением акцента с программирования на проектирование и анализ, а также на вопросы распределенных систем и стандартов. К тому же внимание стало уделяться приложениям, требующим серьезных вычислений и обработки сложных данных. Это привело к появлению объектно-ориентированных баз данных. Практически все коммерческие версии программных продуктов созданы на основе объектно-ориентированных методов. Во многих современных проектах по разработке программного обеспечения среднего уровня сложности, а также в больших и важных проектах уже используют объектно-ориентированное программирование и соответствующие методы. В главах 6–9 будет глубже рассмотрена разработка объектно-ориентированного программного обеспечения. В главе 10 описывается несколько приложений объектно-ориентированного подхода и программирования. В табл. 1.1 подводятся итоги краткого описания истории первых трех десятилетий развития объектной технологии.

Таблица 1.1. Три десятилетия развития объектно-ориентированных методов

Фаза I: 1971–1980 гг.	Фаза II: 1981–1990 гг.	Фаза III: 1991–2000 гг.
Период изобретения	**Период замешательства**	**Период созревания**
Моделирование дискретных событий	WIMP-интерфейсы Xerox & Apple	Акцент на анализ, проектирование, архитектуру и бизнес-модели
Язык Simula	Расширения Lisp	Коммерческие приложения
Kay: машина Flex	Среда ИИ	Распределенные и Web-системы
PARC: Dynabook	Новые языки: Eiffel, C++, ...	Объектно-ориентированные базы данных
Smalltalk		Стандарты, шаблоны, Java, компоненты, миграция

Объектно-ориентированные методы являются частью культуры разработки программного обеспечения и все еще остаются незамеченными многими основными компаниями-разработчиками. Со смещением фокуса разработки в область распределенных систем объектная концепция стала наиболее подходящей парадигмой разработки, акцентирующей внимание на инкапсуляции и передаче сообщений. Возрастающее значение стоимости поддержки системы может привести к осознанию, что возможность повторного использования — это *ключевой* вопрос программирования, проектирования и анализа. Однако это все кажется слишком простым и редко используется в спорах между сторонниками и противниками объектно-ориентированного подхода. Только несколько коммерческих проектов, которые используют объектно-ориентированные технологии, обеспечивают достаточную степень повторного использования кода. Участники лишь некоторых из них, как описывается в главе 2, сообщали о таком преимуществе. Однако развивающийся рынок компонентов требует повторного использования в довольно больших масштабах. Большинство экспертов верят, что для успеха CBD требуется больше внимания уделять архитектуре программного обеспечения и увеличению роли моделирования объектов. К тому же с уходом в историю проблемы 2000 года большие организации возвращаются к истокам и переходят к использованию согласованных процессов программной инженерии. Можно надеяться, что это послужит отличительным признаком первого десятилетия двадцать первого столетия. Эти годы будут отмечены как золотой век архитектуры и совершенствования процесса. Более подробно это показано в табл. 1.2.

Таблица 1.2. Непосредственное будущее объектно-ориентированных методов

Фаза IV: 2001–2010 гг.
Золотой век архитектуры и совершенствования процесса
Концентрация внимания на архитектуре и шаблонах (микроархитектура)
Формирование ОО процессов разработки
Широкое распространение распределенных систем
Переход к компонентно-организованным системам и оболочкам для ранее созданных систем
Компонентно-ориентированная разработка обеспечит настоящее повторное использование
Особое внимание к моделированию бизнес-процессов и инженерии требований
Смещение от C++ к Java и другим безопасным языкам программирования

С моей точки зрения в недалеком будущем будут необходимы не только лучшие, более чистые и эффективные объектно-ориентированные языки, но и лучшие методы для проектирования объектно-ориентированного программного обеспечения и инженерии требований. Результативная смесь языков, такая как C++, с соответствующими инструментальными средствами UI, такими как Microsoft Visual Studio, уже существует и обеспечивает создание множества полезных объектных библиотек низкого уровня. Чтобы сделать эти библиотеки применимыми для коммерческого использования, требуются соответствующие методы и

средства. Есть твердая уверенность, что это — важное предварительное условие для широкого принятия объектно-ориентированного подхода. UML будет основой для этой работы, но он должен развиваться, чтобы отражать реальные потребности бизнеса. Кроме того, необходимы лучшие методы инженерии требований. Методы и модели процессов, связанные с этим вопросом, будут обсуждаться в главах 8 и 9, где основное внимание будет обращено на раннее тестирование, которое должно получить широкое распространение. Конечно, самые полезные библиотеки компонентов будут разрабатываться вместе с реальными программами. Преимущества и потенциальные ловушки объектно-ориентированного подхода проанализированы в главе 2, а в этой главе читатель ознакомится с необходимыми понятиями и терминологией.

Объектная технология подвергается регуляризации, подобно экспертным системам и технологиям, основанным на правилах. Объектно-ориентированные методы составляют часть основных инструментальных средств разработчика программного обеспечения, в состав которых входят языки четвертого поколения 4GL, базы данных, графическое программное обеспечение и т.д. Однако объектная технология все еще впитывает новые идеи, связанные с человеческим фактором, моделированием данных, искусственным интеллектом и другими областями компьютерных наук. Исследования в данной области продолжаются. Однако если брать в целом, объектно-ориентированное программирование можно рассматривать как сформировавшуюся дисциплину, постоянно используемую коммерческими организациями. Технологии объектно-ориентированных баз данных и брокеры объектных запросов также могут рассматриваться как относительно сформировавшиеся области, как будет показано далее в этой главе. Однако если говорить о несовершенстве данной технологии, то в первую очередь следует обратить внимание на область методов. Чтобы понять, почему, надо ознакомиться с основными принципами объектного подхода, чем мы и займемся далее.

1.2. Что такое объектно-ориентированные методы

Как уже говорилось, понятие *объектно-ориентированные методы* является очень обширным, как, собственно, и "объектно-ориентированный" (ОО) и "объектная технология" (ОТ). В частности, оно означает объектно-ориентированное программирование, проектирование, анализ и базы данных, т.е. фактически целую философию разработки систем и представления знаний на базе мощного подхода.

Исторически, как было показано ранее, развитие этой области началось с объектно-ориентированного программирования, и только совсем недавно появился большой интерес к другим вопросам. С административной точки зрения вопросы программирования, возможно, являются наименее интересными, но, чтобы понять основные концепции и терминологию, мы начнем с обзора объектно-ориентированного программирования и связанной с ним терминологии и не будем возвращаться к данному вопросу. Далее, особенно в главах 6, 7 (в которых рассматриваются методы, CASE-средства и сопутствующие вопросы) и 9, будет показано, как эти основные концепции применяются в жизненном цикле разработки системы в качестве метода анализа. Другими словами, мы начнем с конкретных вопросов программирования, а затем перейдем к абстрактным методам проектирования и анализа, после чего рассмотрим вопросы управления процессом. Объектно-ориентированная разработка во многом сводится к так называемой компонентной разработке, хотя, строго говоря, можно создавать компоненты и без использования объектно-ориентированного программирования. Просто готовый

продукт может напоминать объекты своим поведением. Мы разберемся с этим в главе 7. Для меня объектная технология — это нечто большее, чем программирование и даже проектирование. Она обеспечивает общий подход к представлению знаний, который можно применять как для бизнес-моделирования, так и для построения систем. Это будет рассмотрено позже в главах 6–8.

В следующем разделе вводится терминология объектно-ориентированных методов, которая включена в словарь терминов, где также указаны определения некоторых других, возможно незнакомых, терминов. Читатель должен сознавать, что различные авторы иногда используют эти термины совершенно по-разному. Это не удивительно, поскольку данный подход вызывает очень живой интерес. К счастью, в значительной степени благодаря усилиям Object Management Group, появилась некоторая стандартная терминология, которая и будет использоваться в этой книге. Использованные в этой книге термины согласованы, что, в свою очередь, наводит на мысль, что такие сферы, как моделирование бизнес-процессов, искусственный интеллект, семантическое моделирование данных, управление знаниями и объектно-ориентированный подход, рано или поздно должны объединиться.

1.3. Основная терминология и идеи

В следующей главе будет показано, что на изменение структур данных приходится порядка 16% расходов в области информационных технологий. Чтобы понять основы объектной технологии (ОТ), попытаемся разобраться, почему это происходит в традиционных компьютерных системах и как, при надлежащем применении, ОТ помогает снизить эти расходы. В дальнейшем это послужит основой для понимания основных терминов.

Традиционная компьютерная система, основанная на так называемой аппаратной архитектуре фон Неймана (Von Neumann architecture), может рассматриваться как множество функций или процессов, работающих с набором данных, хранимых либо в памяти, либо на диске — это не принципиально. Эта статическая архитектурная модель показана на рис. 1.1. Из рисунка видно, что в процессе работы системы динамика состоит в вызове некоторой функции f(1), которая считывает соответствующие данные A, преобразовывает их и записывает в B. Потом вызывается некоторая другая функция f(2), которая тоже считывает некоторые данные (возможно, те же), использует их по назначению и записывает в C. Подобный перекрывающийся доступ к данным порождает проблемы параллельной обработки и целостности, которые могут быть разрешены с помощью систем управления базами данных. Перед тем как двигаться дальше, стоит рассмотреть вопрос, что нужно сделать, чтобы изменить часть структуры данных.

Рассматривая это с точки зрения специалиста по поддержке программы, можно сделать единственный вывод: необходимо проверить, затрагивают ли эти изменения каждую отдельную функцию. В этом вопросе может помочь качественная документация, но на практике она доступна редко. Отчасти это объясняется тем, что хорошая документация сама должна состоять из объектно-ориентированного описания системы, и ее невозможно представить в отрыве от объектно-ориентированной реализации или, по крайней мере, проектного решения. Кроме того, изменение каждой функции в соответствии с новыми структурами данных может иметь побочный эффект в других частях системы. Возможно, это объясняет необычайно высокую стоимость поддержки программных систем.

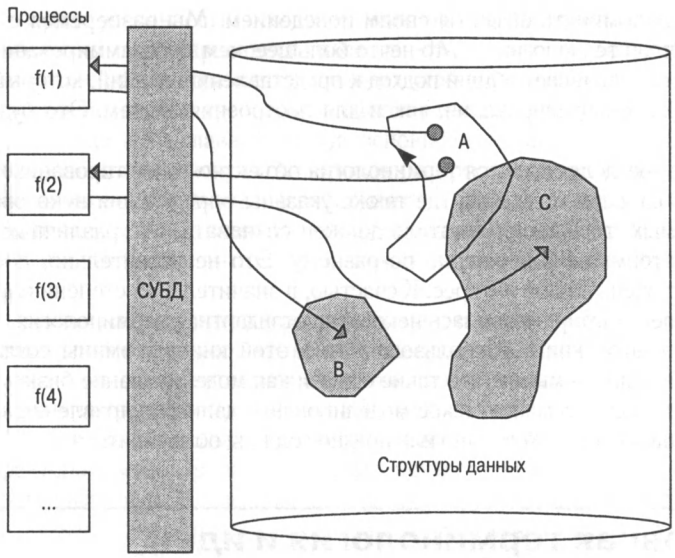


Рис. 1.1. Архитектура обычной компьютерной системы

На рис. 1.2 показан совершенно иной архитектурный подход к системам. Все данные, доступ к которым нужен функции, инкапсулируются в один пакет с этой функцией — так называемый *объект* (object) — таким образом, чтобы другой объект не имел доступа к этим данным. Используя аналогию, предложенную Стивом Куком (Steve Cook), можно рассматривать эти объекты как яйца. Желток — это структура данных, белки состоят из функций, которые имеют доступ к этим данным, а скорлупа представляет сигнатуру общедоступных операций. Оболочка интерфейса скрывает реализацию и самих функций, и структур данных. Предположим, что в яйце, изображенном "в разрезе" на рис. 1.2, изменилась структура данных. Тогда специалисты по сопровождению должны проверить только факт влияния изменений на белок этого яйца; сфера вмешательства локализована. Изменение реализации одного объекта не может затронуть "внутренность" другого. В этом и состоит **инкапсуляция**: данные и процессы **объединяются** и **скрываются** за интерфейсом.

Однако при использовании этой модели в чистом виде возникает проблема. Допустим, каждый объект содержит функции, которым нужны одинаковые данные. В таком случае появляется необходимость дублирования данных, и подход становится довольно непрактичным. Решение состоит в передаче сообщений между объектами. Тогда объект X сможет использовать данные A, но не инкапсулировать их. При условии, что в желтке X хранится идентичность другого объекта Y, который содержит необходимые данные, он может передать сообщение, запрашивающее эти данные или даже несколько преобразованную их версию. Это показано на рис. 1.3, где маленькая черная точка символизирует идентичность целевого объекта, а стрелки показывают направление передачи сообщения. Можно сказать, что это — половина принципа объектной технологии. Вторая половина — это возможность классификации объектов и их связывания различными способами. Обратите внимание на то, что этот подход локализует и таким образом сильно упрощает исходную задачу.

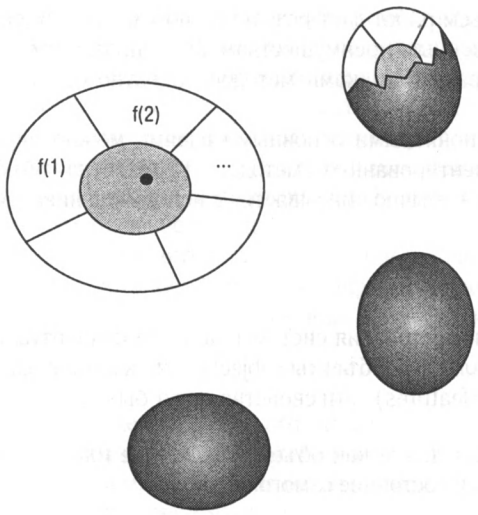


Рис. 1.2. Архитектура объектно-ориентированной системы

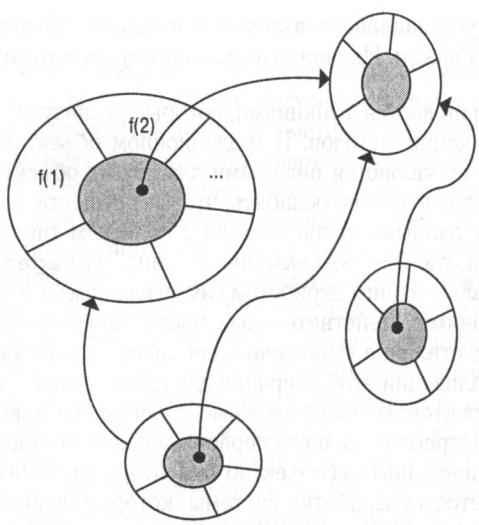


Рис. 1.3. Передача сообщений исключает дублирование данных

При изменении структуры данных программисту нужно всего лишь проверить функции в белке яйца, инкапсулирующего эти данные. Это изменение не может повлиять на другие части системы, если оболочка не повреждается или не деформируется, т.е. если не меняется интерфейс. Таким образом, если необходимо на порядок уменьшить проблемы сопровождения, нужно очень упорно поработать, чтобы обеспечить корректные, полные и устойчивые интерфейсы объектов. Из этого следует, что в данном случае согласованный анализ и проектирование еще более эффективны и необходимы, чем для традиционных систем.

Этот дополнительный объем работ дает результат, потому что объектная технология ведет к некоторым очень существенным преимуществам. Любопытно, что этот принцип инкапсуляции часто игнорируется разработчиками методов объектно-ориентированного анализа, как будет показано позже.

С этими несомненно понятными основными идеями можно переходить к расшифровке "жаргона" объектно-ориентированных методов. Разработка объектно-ориентированного программного обеспечения обычно описывается с использованием указанных ниже терминов и концепций.

Объекты

Основными единицами построения системы на этапе концептуализации, проектирования или программирования являются **объекты** (object) или экземпляры, организованные в классы с общими **свойствами** (features). Эти свойства могут быть трех видов.

- **Атрибуты** (attribute), такие как объем, положение или цвет, символизируют связи с другими объектами и состояние самого объекта.
- Процедуры или услуги, предоставляемые объектом, такие как перемещение или расширение. Их называют **операциями** (operation) или **методами** (method).
- Правила, которые устанавливают взаимосвязи свойств объекта или определяют условия его жизнеспособности. Их иногда называют **инвариантами** (invariant).

Строго говоря, методы являются функциями, которые реализуют операции, а операции — это абстрактные спецификации методов. В традиционном объектно-ориентированном программировании атрибуты не являются видимыми для других объектов, но ниже и в главе 6 будет показано, что это условие можно ослабить, что даст существенные преимущества. Идея пакетирования функций с данными тесно связана с понятием типа данных в традиционных языках программирования, где 3 — это экземпляр "типа" `integer` (целое число), а целые числа характеризуются только одним атрибутом (их значением) и несколькими допустимыми арифметическими операциями. Действительные числа характеризуются похожими операциями, но реальное осуществление умножения для чисел с плавающей точкой совершенно отличается от способа реализации этой операции для целых чисел. Аналогично задавая параметры и методы, можно трактовать более сложные объекты, часто встречающиеся в моделях данных. Слово "метод" употребляется в его первоначальном значении, пришедшем из языка Smalltalk, и никак не связано с иными (объектно-ориентированными) методами. В последнем случае так называются методы разработки системы, которые подробнее обсуждаются в главах 6–8. В данном же контексте **метод** (method) определяется как процедура или функция, которая изменяет состояние объекта или заставляет объект отправить сообщение. Описание или сигнатура метода называется **операцией** (operation). Операции показывают, какие сообщения объект может успешно обрабатывать.

Класс (class), в смысле объектно-ориентированного программирования, — это совокупность объектов, которые имеют общие свойства и методы. В этой книге термин "класс" обычно используется во множественном числе. Класс может рассматриваться как шаблон для построения экземпляров. **Тип** (type) объекта — это спецификация класса, а класс — это реализация типа. Тип объекта отражает идею (*содержание* класса), а не коллекцию свойств (*расширение*), поэтому имеет уникальное название. Атрибуты и методы типа объекта часто

рассматриваются как его **свойства** (features) или **обязанности** (responsibilities). Атрибут представляет обязанность знания чего-либо, а метод — обязанность выполнения.

По мере возможности объекты должны основываться на реальных сущностях и понятиях приложения или предметной области. Объекты могут быть или классами, или экземплярами, хотя некоторые авторы и стандарты, такие как UML, используют термин *объект* как синоним слова *экземпляр*. Как отмечено в работе [71], в некоторых языках класс может быть экземпляром класса более высокого уровня или метакласса. В этой книге везде будет использоваться (несколько жаргонное) значение термина *объект* для обозначения класса или экземпляра. Точные термины будут применяться только там, где различие существенно. Термин *объект* можно четко сформулировать как "нечто идентифицируемое", но на этом внимание будет остановлено не раньше главы 5, которая посвящена обсуждению именно этого вопроса.

Идентичность

На уровне концептуального моделирования объекты (типы, классы или экземпляры) имеют уникальную идентичность на протяжении всей своей жизни. Это отличает объектно-ориентированные модели от, скажем, реляционных. Это чрезвычайно важно в объектно-ориентированных базах данных. На уровне языка программирования можно возразить, что класс не имеет идентичности, но пока этот вопрос обсуждаться не будет.

Инкапсуляция

Структуры данных и элементы реализации метода объекта являются невидимыми для других объектов в системе. Единственный путь получения доступа к состоянию объекта — это передача сообщения, вызывающего один из его методов. Строго говоря, доступ к атрибутам осуществляется через методы, которые считывают и устанавливают их значения. Другими словами, атрибуты — это словарь, с помощью которого можно обсуждать видимое извне поведение объекта. Вообще говоря, в программировании это обеспечивает эквивалентность объектного и абстрактного типов данных. Однако для полного описания объекта необходимо также четко определить его интерфейс. Некоторые методы объекта могут быть скрыты за интерфейсом — это **закрытые** (private) методы. Интерфейс лучше всего рассматривать как общедоступное описание **обязанностей** (responsibility) объекта. Атрибуты могут быть рассмотрены как *обязанность знания*, а операции — как *обязанность действия*. Другими словами, можно сказать, что общедоступный интерфейс определяет *вопросы, которые можно задать объекту*, и *программы действия, которые можно ему предложить*. Такое представление объекта — иногда его называют "очеловечиванием" — позволяет рассматривать данный объект как маленького искусственного человечка, способного поддерживать диалог с другими особями и размышлять о собственных характеристиках. Это представление очень удобно на этапе выделения требований и анализа систем.

Сообщения

Объекты (классы и их экземпляры) общаются посредством передачи сообщений. Это, по большей части, исключает дублирование данных и гарантирует, что изменение структур данных и процедур, инкапсулированных в пределах объектов, не распространяет свое влияние на другие части системы. Сообщения реализуются как вызовы функций. Сообщения всегда возвращают данные, отправленные объектом. Объект может отправить сообщение другому объекту только в том случае, если в нем хранится идентичность другого объекта. Это

может рассматриваться как недостаток объектно-ориентированного модельного представления, когда существует необходимость передачи сообщений многим объектам. Однако позже будет показано, что существуют пути устранения этой проблемы.

Наследование

Причину важности наследования можно понять, рассмотрев аналогию с вареным яйцом. Выше было показано, что сопровождение системы может быть упрощено, а необходимые изменения локализованы, потому что реализация скрыта от других объектов за интерфейсом. Следовательно, это преимущество предполагает, что интерфейс никогда не изменится. Однако наш мир не идеален. Люди — даже разработчики — могут ошибаться. Технические требования меняются. Таким образом, интерфейс может подвергаться изменениям и (в связи с обновлением технологии) даже большим, чем при традиционных методах разработки. Ответ — это издание законов, запрещающих изменение интерфейсов и настойчиво требующих, чтобы видоизменения затрагивали только подклассы, которые расширяют или, возможно, заменяют свойства имеющихся объектов. Любое отклонение от этого режима должно рассматриваться как базовая перестройка архитектуры системы. Это правило предполагает, что классы предназначены для расширения, и укрепляет позиции объектно-ориентированного анализа и проектирования. Экземпляры (обычно) наследуют все свойства классов (и только их), которым они принадлежат. Это называется **классификацией** (classification). Но в объектно-ориентированной системе можно дать возможность классам наследовать свойства более общих суперклассов. В этом случае унаследованные свойства могут перекрываться, кроме того, для обработки исключительных ситуаций могут вводиться дополнительные свойства. Наследование реализует идеи **специализации** (specialization) и **абстракции** (abstraction) и представляет частный случай структурной взаимосвязи между группами классов. Наследование — это только одна из абстрактных структур, с помощью которой мы упорядочиваем мир; но она чрезвычайно важна, и ее важность соответствует важности глагола *to be* в английском языке. Позже будет рассмотрена еще одна ключевая структура — композиция (соответствующая глаголу *to have*).

Полиморфизм

Возможность использовать одинаковые выражения для обозначения разных операций называется полиморфизмом. Это, например, использование знака + для обозначения сложения в классе вещественных или целых чисел, применение сообщения "add 1" ("прибавить 1") и к счету в банке, и к списку памяток: похожие сообщения дают совершенно разные результаты. Полиморфизм обеспечивает возможность абстрагирования общих свойств. Он часто реализуется через *динамическое связывание* (dynamic binding). Наследование — это частный случай полиморфизма, который характеризует объектно-ориентированные системы. Некоторые специалисты утверждают, что полиморфизм является *центральным* понятием в объектно-ориентированных системах, но существуют не объектно-ориентированные модельные представления, в которых тоже присутствует данное понятие. Это можно видеть на примере таких языков, как ML или Miranda.

Подстановки

Чтобы воспользоваться преимуществами повторного использования программного обеспечения, должна существовать возможность замены родительских классов их подклассами. Например, если для представления французов создается новый подкласс класса `People`

(люди), то любое сообщение, которое "понимает" класс `People`, должно быть понятно новому подклассу. Из этого следуют определенные проектные ограничения, которые на уровне абстрактного моделирования часто идут вразрез со здравым смыслом: например, задание многоугольников как подкласса прямоугольников. Это будет обсуждаться позже, в главах 6 и 9.

Делегирование: бесклассовое наследование

Объектная технология является **классово-ориентированной** (class-oriented) в том смысле, что экземпляры извлекают свои свойства из классов, которые в свою очередь могут извлекать свои свойства из более абстрактных классов, находящихся выше по иерархии, или из группы классов. Еще один способ достижения преимуществ объектно-ориентированного подхода — использование бесклассовой парадигмы, где каждый объект рассматривается как прототип в следующем смысле. Каждый объект — это экземпляр, который рассматривается как типичный. Другие экземпляры могут быть порождены путем незначительных изменений свойств существующих экземпляров. Таким образом, стандартная собака (по кличке Шарик или Дружок) имеет четыре лапы. Существует несколько экземпляров собак (с другими именами), порожденных на основе шаблона или прототипа, определенного собакой Шарик. Пес моего друга (по имени Спок) потерял лапу в результате несчастного случая. Таким образом, эта собака полностью подобна Шарику, за исключением того, что имеет три лапы. Языки, поддерживающие эту модель наследования, такие как SELF [767], называются **языками прототипов** (prototype language). Говорят, что они поддерживают бесклассовое **наследование** или **делегирование** (delegation). Фреймы и сценарии ИИ тоже могут рассматриваться как прототипы, хотя они также позволяют использовать наследование классов. **Системы на основе исполнителей** также используют идею передачи (делегирования) функций. Они позволяют объектам передавать другим объектам разрешение выполнять операции от их имени. Языки реализации таких систем — это обычно языки очень низкого уровня, по сравнению с языками объектно-ориентированного программирования.

В двух словах

Итак, структуры данных и детали реализации объекта являются невидимыми для других объектов в системе. Единственный способ получения доступа к состоянию объекта — передать сообщение, которое инициирует выполнение некоторого метода. Интерфейс можно рассматривать как общедоступное объявление *обязанностей* объекта. В целом, реализация объектно-ориентированного подхода в языке программирования, проектном решении или в компьютерной системе характеризуется двумя ключевыми свойствами.

- Инкапсуляция
- Наследование

Как всегда, когда мы имеем дело с дихотомией, инкапсуляция и наследование не являются четко разделенными понятиями; между этими абстракциями существует некоторое противостояние и взаимное притяжение. Наследование может нарушать принцип повторного использования, потому что подобъекты могут иметь привилегированный доступ к реализации методов других подобъектов. Таким образом, некоторые термины, такие как полиморфизм, можно отнести к обеим категориям, а наследование можно даже рассматривать как форму абстракции. Общие и отличные свойства абстракции и наследования рассмотрены в разделе 1.3.3.

Другие авторы при описании общей концепции предпочитают использовать термин "абстракция", а не "инкапсуляция". В данной книге эти термины взаимозаменяемы, и трудно сказать, какая точка зрения является более предпочтительной[3]. Оба слова включают множество важных концепций. Стоит помнить, что инкапсуляция — это только один способ поддержки принципа сокрытия информации.

В двух последующих разделах рассматривается значение этих двух основных принципов и вводится терминология для дальнейшего использования.

1.3.1. Абстракция и инкапсуляция

Абстракция (которая является ключевым средством инженерии программного обеспечения) включает различные вопросы, и ее трудно охватить целиком и сразу. Часто говорят, что абстракция — это *критический* инструмент объектно-ориентированного проектирования. Наиболее подходящим по смыслу является описание, данное в оксфордском словаре английского языка (Oxford English Dictionary — OED): "результат мысленного отделения". Еще одно близкое определение: "представление основных свойств чего-либо без упоминания предпосылок или несущественных подробностей". С понятием абстракции тесно связана идея полноты в том смысле, что она должна инкапсулировать *все* существенные свойства предмета. В объектно-ориентированном программировании этот термин обозначает, что объекты должны абстрагировать и инкапсулировать как данные, так и процессы. Познание осуществляется не только через свойства, но и через поведение: *по делам узнаю я их*. Абстракция является инкапсулированной, если она состоит из интерфейса, видимого для внешнего мира, и невидимой реализации. Интерфейс может рассматриваться либо как общедоступное "лицо" класса, либо как нечто обособленное от него, которое определяет, что реализует класс.

Одно важное различие между стандартными типами и интерфейсами, типами и классами в объектной технологии состоит в том, что последние не полностью определяются атрибутами и операциями (сигнатурой типа). Объектные интерфейсы могут также содержать **описание семантики** (assertion): утверждения о сигнатуре. Характерный элемент семантики — это пред- или постусловие метода или инвариант класса. Это будет очень важным при изучении объектно-ориентированного анализа и проектирования в главе 6, но в данный момент мы не будем акцентировать внимание на данном вопросе.

Абстрактный тип данных, АТД (abstract data type — ADT), — это абстракция (подобная классу), которая описывает совокупность объектов в терминах инкапсулированных или скрытых структур данных и операций над этими структурами. Абстрактные типы данных, в отличие от базовых, подобных `Integer`, могут быть описаны пользователем в создаваемом приложении, а не разработчиками базового языка программирования. Следует отметить, что идея абстрактных типов данных очень напоминает "типы объектов", используемые в методах моделирования данных. Это не случайно. На этом строится материал главы 6. Однако ключевая разница между АТД или классами в объектно-ориентированном программировании и типами объектов состоит в том, что АТД содержат в себе методы. Например, абстрактный тип, представляющий длину в неметрической системе единиц, должен включать методы сложения футов и дюймов.

---

[3] Подробнее см. [71].

На рис. 1.4, *а* показан класс, называемый `Employees`, вместе с некоторыми его атрибутами и методами. Однажды описанная, эта абстракция доступна разработчику непосредственно и просто. На рис. 1.4, *б* показан этот же тип в более ранней объектно-ориентированной системе обозначений, в которой не учитываются переменные-члены экземпляра. В этой книге будут использованы обозначения UML, которые показаны на рис. 1.4, *а*.

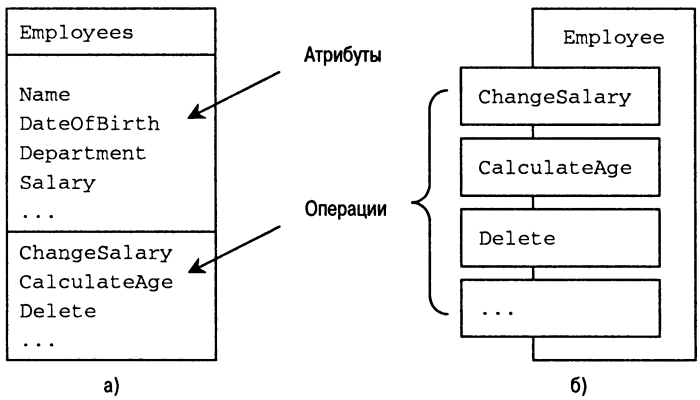


Рис. 1.4. Класс `Employees`, описывающий понятие сотрудника в системе обозначений UML (а), и объект типа `Employee` в другой простой системе обозначений, не учитывающей атрибуты (б)

С точки зрения программиста между классами и типами существуют отличия, потому что информация о типе — это только спецификация объекта; класс, которому он принадлежит, может быть определен только во время выполнения программы. Различие состоит в том, что классы описывают спецификации, которые могут совместно использоваться совокупностями экземпляров или другими классами, а не только экземплярами. Однако с точки зрения аналитика классы и типы — по сути, одно и то же.

Выше было указано, что в объектно-ориентированном программировании принято считать, что уникальная идентичность присуща только экземплярам, что отражает инстанцирование экземпляров в компьютерной системе во время выполнения. Вне контекста программирования это является ошибкой, поскольку класс также имеет уникальную идентичность. Как-никак, в концептуальном мире существует только один класс `Apple`. В этой книге **объект** — это нечто, имеющее уникальную идентичность. Он может быть конкретным субъектом, реальным или вымышленным, концепцией, абстракцией или чем-то конкретным. Это противоречит многим определениям, содержащимся в книгах и статьях, где термин "объект" используется как синоним слова "экземпляр". Поскольку обсуждается объектная технология, "объект" должен представлять собой общий термин в пределах предметной области. Таким образом, если имеется в виду класс, то слово "объект" можно рассматривать просто как сокращенную форму термина "тип объекта".

Абстрактные классы (abstract class) не могут иметь экземпляров; они существуют только для обеспечения основных концепций, которые будут использоваться подклассами. Когда различие между классом и экземпляром становится значительным, термин "объект" уже нельзя использовать. **Реальные классы** (concrete class) могут иметь экземпляры.

Объекты имеют два аспекта: внутренний и внешний. Внутренний аспект описывает состояние, реализацию и инстанцирование объекта. Внешне представление, в чисто объектно-ориентированном стиле, отражает только имена методов и типы их параметров. Оно показывает, что может делать объект (как на рис. 1.4, *б*). Автор немного подкорректировал эту "чистую" точку зрения, показав атрибуты объектов, как на рис. 1.4, *а*. Чтобы вернуться к чистой точке зрения, можно идентифицировать эти атрибуты не с внутренним состоянием, а со стандартными методами, которые обеспечивают доступ и обновление. Однако если это не приводит к путанице, атрибуты удобно рассматривать непосредственно. Во многих книгах и статьях по объектно-ориентированному программированию на такой точке зрения настаивают очень строго. Однако приведенные примеры обычно описывают программные абстракции очень низкого уровня, такие как множества, множества с повторяющимися элементами, коллекции, стеки и т.д. Например, стек описывается как структура данных с четырьмя методами: добавление `push`, выталкивание данных `pop`, проверка на пустоту `empty` и получение верхушки стека `top`. Это говорит о том, что стек — это упорядоченный список данных, элементы которого добавляются и удаляются только с одного конца (рис. 1.5). Доступ к данным стека осуществляется только через эти методы, и нет необходимости описывать их реализацию. Стек может быть реализован как связный список или массив с указателем на верхушку стека. Стек не имеет видимых атрибутов, хотя в качестве таковых могут рассматриваться методы `top` и `empty`.

В коммерческих системах объекты гораздо сложнее, и невозможно представить себе абстрактное понятие сотрудника или счета без описания их атрибутов. Приверженцы старых подходов могут продолжать представлять каждый атрибут А парой стандартных методов доступа: получения и установки значения А.

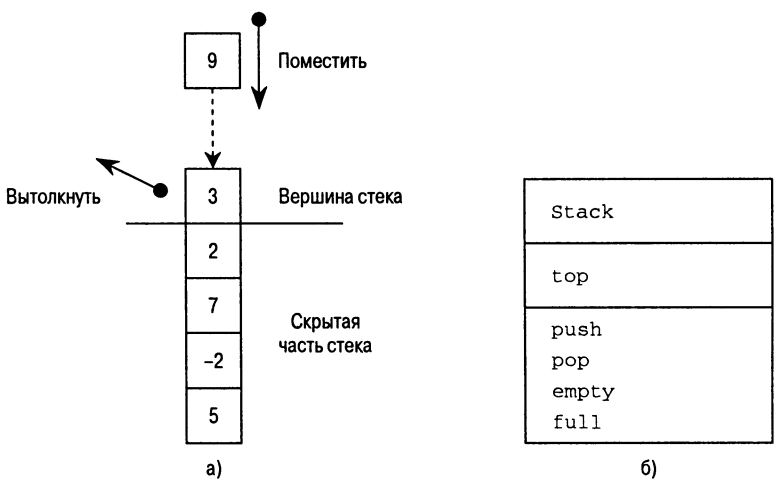


Рис. 1.5. Физическое представление конкретного стека целых чисел (а) и протокол объектного типа стека на языке UML (б)

Экземпляры объектов (строго говоря, классов или "объектных типов") аналогичны записям в базе данных. Они содержат конкретные данные и обладают свойствами объекта. Все экземпляры объекта имеют одинаковое множество атрибутов и методов. Это может не выполняться для более общих классов, потому что наследование класса обеспечивает возможность

специализации классов с удалением или добавлением дополнительных атрибутов и методов. Например, стек — это специальный вид списка, имеющий множество методов конкатенации. Еще одним примером может быть домашняя кошка, которая наследует свойства кошки в целом, но дополнительно имеет хозяина.

Атрибуты экземпляра иногда называются **переменными экземпляра** (instance variable), а атрибуты класса — **переменными класса** (class variable). Переменные класса совместно используются *всеми* экземплярами класса. Например, число экземпляров класса — это подходящий кандидат для переменной класса. Кроме того, различают **методы класса** (class method) и **методы экземпляра** (instance method). Методом класса может быть вычисление суммы или среднего значения некоторого атрибута среди всех экземпляров класса.

Под общим понятием абстракции понимают много малопонятных технических терминов, таких как инкапсуляция, полиморфизм и обобщение. Рассмотрим смысл некоторых из этих терминов. Поступая таким образом, мы обнаружим новые основные свойства объектно-ориентированного стиля.

Инкапсуляцией (которая поддерживает принцип **сокрытия информации**) называется включение в объект всей необходимой ему информации таким образом, чтобы другим объектам не требовалось знаний об этой внутренней структуре. Так, в примере, показанном на рис. 1.4, детали алгоритма изменения оклада `ChangeSalary` могут быть скрыты внутри объекта `Employees`, при этом другие объекты или даже пользователи не смогут получить эту информацию. Подобным образом можно использовать закрытые данные объекта, такие как зарплата. Реализация хранилища данных тоже всегда должна быть закрытой в рамках объекта. Для доступа к данным о зарплате (или для их изменения) не обязательно знать, в каком виде они хранятся: как целое число или число с плавающей точкой. То же можно сказать и о методах; их детальная реализация должна быть скрыта от других объектов, а видимым должно быть только поведение. Помимо идентичности, объекты могут иметь внутреннее состояние, но оно не должно быть доступно напрямую. Одно из следствий этого принципа — клиенты объектов не подвергаются опасности при изменении реализации, если при этом не меняется интерфейс. Невидимые или инкапсулированные части объекта — это его **закрытая реализация** (private implementation); упомянутые выше видимые атрибуты и методы составляют **открытый интерфейс** (public interface) объекта.

В [71] предлагаемый в данной книге подход (как и в работах множества других авторов) подвергается критике за смешение понятий абстракции, инкапсуляции и сокрытия информации. Впрочем, автор это делал умышленно. Сокрытие информации [617] — это принцип модульности и закрытости проектного решения одних объектов для других. Берард определил *абстракцию* как процесс, посредством которого мы решаем, какая информация должна быть видимой и какая нет. Инкапсуляция — это только стратегия упаковки, используемая для реализации этих решений. Он утверждает, что сокрытие информации — это не обязательно хорошо, и, конечно, это не уникальная идея для объектно-ориентированного программирования. Хотя я до некоторой степени и поддерживаю этот подход, все же хочу заметить (в чем убежден), что смешение этих понятий во вводном тексте не создает путаницу, а облегчает понимание.

Стратегии связывания

В программировании возможность определять класс объекта во время выполнения программы и выделять для него память называется **динамическим связыванием** (dynamic binding). В языках, поддерживающих статическое связывание, память под объекты и их типы

распределяет компилятор, определяя их класс раз и навсегда. Поскольку эта книга посвящена далеко не только программированию, различия между понятиями класса и типа учитываться на будут. Однако важно помнить, что класс — это не просто множество. Класс включает переменные-члены и действия, а множество — только члены. Динамическое связывание противоположно **раннему** или **статическому связыванию** (static binding), когда распределение памяти для разных типов осуществляется компилятором. Динамическое связывание называют также **поздним** (late binding). **Динамическое связывание** — это методика программирования, которая реализует принцип полиморфизма в языках объектно-ориентированного программирования. Платой за повышение гибкости при динамическом связывании является снижение быстродействия и более низкая производительность, по сравнению со статически связываемыми системами.

На рис. 1.6 показаны способы реализации динамического связывания и полиморфизма.

Снова сообщения

В объектно-ориентированных системах данные извлекаются из объекта одним единственным способом — путем передачи объекту **сообщения** (message). Сообщение содержит адрес (объекта или объектов, которым оно передается) и инструкцию, состоящую из имени метода и списка (возможно, пустого) параметров.

Если адресат содержит метод, для которого инструкция имеет смысл, то ответ возвращается передающему объекту. Таким образом, целочисленному объекту 3 можно передать сообщение `Add(5)` (прибавить 5) и ожидать ответа `8`. В ответ на сообщение "сообщите оклад сотрудника с именем Эрика", возвращается либо запрашиваемая информация, либо что-то похожее на "У вас нет прав доступа к данным об окладе Эрики", если при обращении к некоторой таблице ограниченного доступа используется определенная процедура сокрытия информации. Если адресат не содержит метода, который может обработать сообщение, возвращается стандартное сообщение об ошибке. Например, такая ошибка имела бы место, если сообщение `CalculateAge` (вычислить возраст) передать объекту `3`, экземпляру типа `Integer`. При использовании компилируемого языка программирования, ошибки такого вида должен обнаруживать компилятор еще до того, как в процессе выполнения программы может возникнуть эта ситуация. Напомним, что методы, инкапсулированные внутри объекта, точно определяют, какие сообщения этот объект может успешно обработать. Совокупность сообщений, на которые может отвечать объект, иногда называется **протоколом** (protocol). Имя сообщения иногда называют его **селектором** (selector). Различные получатели могут поразному интерпретировать одно и то же сообщение.

В строго (статически) типизированных языках программирования компилятор гарантирует, что объекты не смогут получать несанкционированные сообщения. При динамическом связывании каждый объект сам отвечает за защиту от "нелегальных" сообщений. Отметим, что сообщения поступают с интерфейса объекта (поскольку реализация является скрытой).

Если классы рассматриваются как компоненты, протокол часто делится на (возможно, перекрывающиеся) множества сообщений, называемые **интерфейсами** (interface). Обычно, однако, под интерфейсом подразумевают полное множество сообщений.

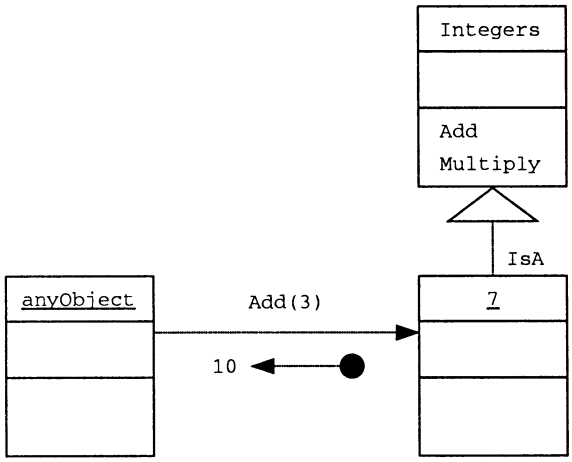


Рис. 1.6, а. Сообщение `Add(3)` передается целому числу 7, которое наследует процедуру сложения от класса целых чисел

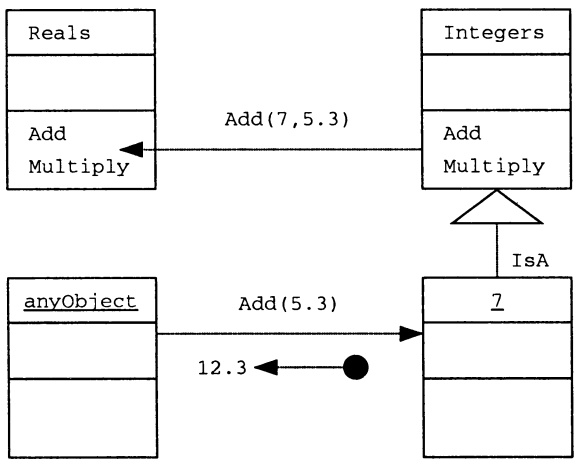


Рис. 1.6, б. Сообщение `Add(5.3)` не может быть обработано в классе целых чисел, поэтому вызывается соответствующий метод класса вещественных чисел. Это называется **перенаправлением**

Множественная абстракция

Итак, термины инкапсуляция, абстракция данных и сокрытие информации означают практически одно и то же. Однако некоторые специалисты различают простое сокрытие информации и так называемую множественную абстракцию. Под этим понимается следующее. Конкретные экземпляры рассматриваются как объекты, принадлежащие множеству с отдельно определенными свойствами и наследующие эти свойства. Например,

Фред — это экземпляр мужчины, у него карие глаза, но он наследует большинство своих свойств от абстрактного множества мужчин, включающего атрибут "цвет глаз".

Еще одна особенность объектов — они уникально идентифицируются на все время жизни. В системах баз данных идентичность объекта порождает некоторые существенные проблемы и обеспечивает некоторые значительные преимущества. Об этом говорится позже, в главе 5.

Еще о полиморфизме

В различных контекстах возможность использовать один и тот же символ для разных целей называется по-разному: **полиморфизм** (polymorphism), **перегрузка** (overloading) и **перегрузка операций** (operator overloading). Например, передача сообщения `Add` целому и действительному числу фактически активизирует совершенно разные процедуры (рис. 1.7), но в обоих случаях удобно использовать одинаковую форму записи +; это облегчает восприятие и делает язык легче для изучения и запоминания. Аналогично команда `delete` может быть необходимой в различных случаях для разных объектов, особенно если при удалении записи из базы данных должен обеспечиваться контроль целостности. Формально, полиморфизм — имеющий много форм — означает возможность переменной или функции принимать разные формы во время выполнения программы или, более точно, возможность обращаться к экземплярам разных классов. Перегрузка — это особый случай, при котором для обозначения двух разных операций просто используется одно имя, как, например, операция `open` (открыть) применяется к файлам и окнам.

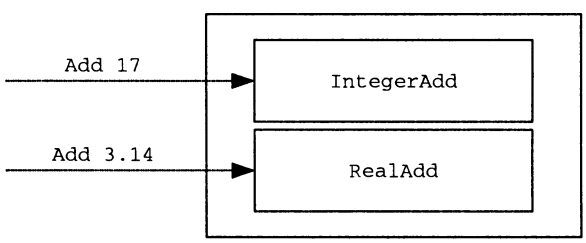


Рис. 1.7. Перегрузка операции (полиморфизм) суммирования

На самом деле в языке Smalltalk понятие полиморфизма используется несколько некорректно. Чистый принцип лучше представлен в функциональных языках, таких как ML, где функции могут иметь аргументы разных типов. Хотя понятие полиморфизма, как оно представлено выше, вполне подходит для последующего изложения материала данной книги, по этому вопросу существует много теоретической литературы, в которой внимание акцентируется на множестве тонких моментов.

В [141] различается несколько видов полиморфизма. На самом верхнем уровне выделяется специальный и универсальный полиморфизм. Специальный полиморфизм — это использование одного и того же символа для семантически несвязанных операций. К этой категории относится перегрузка операций, например использование символа + для суммирования целых чисел и матриц. Еще одна разновидность специального полиморфизма, так называемое приведение, позволяет реализовать операции для входных данных комбинированного типа, например складывать целые числа с вещественными. Универсальный полиморфизм делится на

параметрический и полиморфизм включения (или наследования); последний рассматривается в разделе 1.3.2. Параметрический полиморфизм — возможность выбора аргументов в вызове функции из некоторого диапазона типов. В нашем изложении его можно считать обобщением. Типы полиморфизма представлены в табл. 1.3.

Таблица 1.3. Классификация типов полиморфизма по Вегнеру

Специальный полиморфизм	Универсальный полиморфизм
перегрузка	параметрический
приведение	полиморфизм включения

Обобщение

Возможность определять параметризованные модули называется **обобщением** (genericity); данная возможность используется во многих языках, которые обеспечивают инкапсуляцию, таких как Ada. Пример родового типа — это список, который может быть списком имен, целых чисел или неких специфических объектов, например имен сотрудников (рис. 1.8). Реальный тип определяется только по контексту. В таких языках, как Ada, Modula-2 и даже ALGOL, существует возможность определять параметризованные модули. Обычно параметры являются типами. Родовые сообщения обеспечивают создание повторно используемых компонентов и каркасов за счет устранения зависимости вызова процедур от реального содержимого вызова.

Обобщение и наследование можно рассматривать как *альтернативные* методики создания расширяемых и повторно используемых модулей, но, как будет показано далее, обобщение более ограничено на практике. Тем не менее это свойство полезно иметь в языке программирования, потому что непосредственно повторно используемые компоненты должны применяться к нескольким типам. Например, механизм поиска или сортировки, который работает только с числами, практически бесполезен.

Идентификация объектов

Важный вопрос — как идентифицировать объекты в реальном приложении. Этот вопрос порождает глубокие философские дебаты и часто вызывает большое замешательство. С точки зрения эмпирика, объекты — это нечто, обнаруживаемое органами восприятия, и их выделить очень легко. С феноменологической точки зрения, восприятие — это активный, созидательный процесс, и объекты возникают как из нашего понимания, так и из окружающего мира посредством диалектического процесса. Еще более распространенная точка зрения предполагает, что реальные абстракции — это отражение общественных отношений и человеческой созидательной деятельности, для которой существует объективная основа в реальном мире. Мы вернемся к этой противоречивой задаче в следующих главах. Пока просто определим, что объекты могут быть по-разному представлены аналитиком, знакомым с приложением, и что выбор "лучшего" представления — это фундаментальный вклад человеческого разума в вычислительные системы.

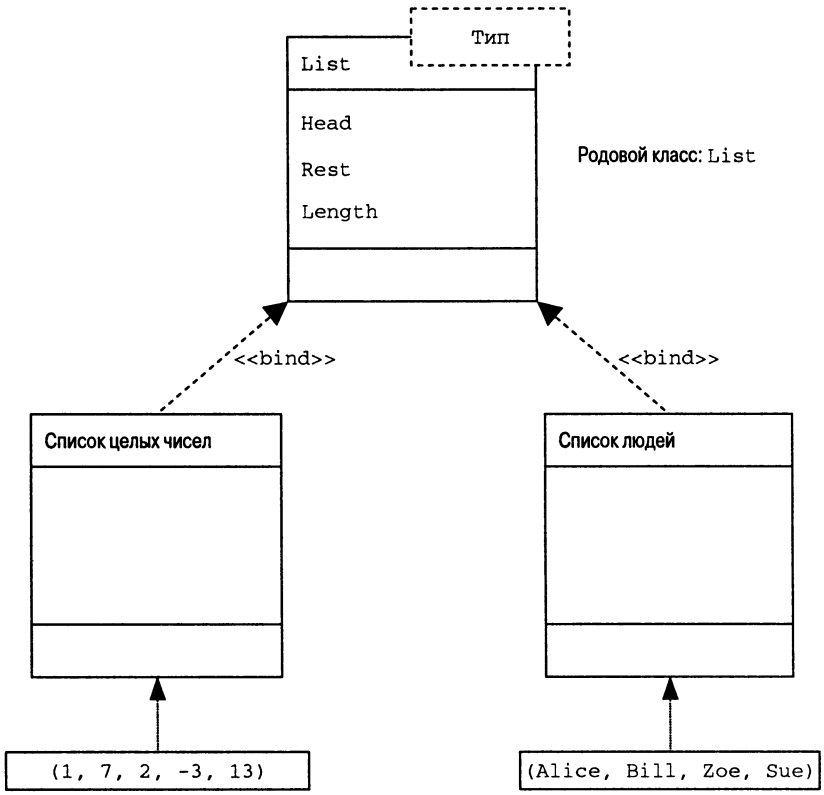


Рис. 1.8. Использование обобщения при моделировании наследования для родового класса `List` (список)

Некоторые объекты соответствуют непосредственно реальным объектам, таким как `Employees`, а другие, такие как стеки, соответствуют придуманным абстракциям. Абстракции отражают реальное разделение предметной области. Уровень абстракции может зависеть от приложения, но это ставит под угрозу повторное использование абстракции, и проектировщик должен определять абстракцию максимально широко. С другой стороны, нужно следить, чтобы не потерять эффективность, создав абсурдные и неоправданные общие объекты. Так, при определении класса `Employees` мы не включаем в число его атрибутов ни "родную планету", ни курсы галактических валют, необходимые для процедуры выдачи заработной платы, хотя некоторые будущие разработчики систем могут упрекнуть нас за такую недальновидность. Что в этом случае должен делать разработчик? Использовать концепцию наследования, к которой мы сейчас и перейдем.

В следующей главе будет показано, что первая причина для выделения абстракции и инкапсуляции — это получение кода, пригодного для повторного использования. Использование объекта посредством его спецификации как абстрактного типа или класса подразумевает, что если его внутренний аспект подвергается изменению, то остальные части системы не будут затронуты. Точно так же, если изменится реализация других объектов системы, то этот

объект тоже не должен быть затронут. Единственная проблема — не должен меняться интерфейс. Таким образом, фиксация "верных" абстракций — это очень важно. Этот вопрос занимает существенное место в объектно-ориентированном анализе и в этой книге.

1.3.2. Наследование

Еще одна характерная особенность объектно-ориентированной системы — это работа со структурными и семантическими отношениями между экземплярами и классами (или типами) и устранение чрезмерной избыточности при хранении одинаковых данных или процедур. Ключевое понятие — это понятие наследования, обобщения или классификационной структуры.

В объектно-ориентированном программировании класс может порождать свои экземпляры в памяти. Эти экземпляры в точности "наследуют" все особенности класса: его методы и атрибуты. Говорят, что класс **классифицирует** его экземпляры, и отношение IsA между экземпляром и его классом называется отношением классификации. Экземпляр может принадлежать только одному классу; другими словами, **множественная классификация** невозможна. Также невозможна и **динамическая классификация**: способность экземпляра менять свой класс во время выполнения программы. Также классы могут наследовать все свойства более общих классов. На рис. 1.9 показан пример ситуации, в которой классы `Cars` (машины) и `Cycles` (велосипеды) являются специализацией (подклассами) класса `Vehicles` (транспортные средства). Для обозначения отношения AKO (A Kind Of — "вид") используется символ UML — маленький равнобедренный треугольник или стрелка. Этот пример иллюстрирует **одиночное наследование** (single inheritance): каждый класс имеет не более одного обобщения. Дальше будет показано, что некоторые языки допускают **множественное наследование** (multiple inheritance), при котором класс может иметь более одного суперкласса. Такие структуры, собственно, называются структурами **обобщения**. Когда речь будет идти о структурах, относящихся к классам и экземплярам, или реализации обобщения и классификации, будем их называть **структурами наследования**. Если экземпляры *в точности* наследуют свойства своего класса, то в подклассах могут добавляться новые свойства. Таким образом, при создании подкласса `MotorizedCycle` (моторизованный велосипед) класса `Cycle` ему необходимо добавить некоторые другие свойства, такие как `FuelType` (тип горючего). Кроме того, в некоторых языках свойства могут перекрываться. Несомненно, такое перекрытие дает возможность подстановки, так что оно запрещено в некоторых языках и проектных решениях.

Если объект имеет тип, тогда он может считаться экземпляром класса. Например, `Fido` — это экземпляр класса собак. Можно пойти дальше и указать, что собака — это частный случай млекопитающего; млекопитающее относится к классу позвоночных животных; и т.д. В итоге получается полная структурная классификация. Формально в число атрибутов каждого экземпляра включается особый атрибут `IsA` (или *слот* в терминологии искусственного интеллекта, где объекты часто называются *фреймами*). Этот атрибут содержит имя класса, которому соответствует экземпляр, или родительский элемент в иерархии. Классы тоже могут иметь родительские элементы, которые будем называть атрибутами **AKO** (A Kind Of). Атрибут AKO содержит список суперклассов, обобщающих рассматриваемый класс. Преимущество такой формализации состоит в том, что класс или объект более низкого уровня может наследовать совместно используемые свойства и методы. При этом исключается необходимость хранения их в каждом экземпляре. Например, атрибут `BirthDate` (дата рождения) совместно используется всеми подтипами класса `Person` (человек); таким образом, нет

необходимости упоминать его явно для класса Employee, если в спецификацию типа включен атрибут (AKO: Person).

Наследуемые методы при определенных обстоятельствах могут перекрываться. Примером может служить метод Delete, унаследованный Employees от базового класса Objects (объекты), содержащего методы для наиболее типичных операций с объектами всех типов. Если поддерживать некоторую безопасность авторизации, для удаления перекрытого стандартного метода может потребоваться особый метод. Хотя перекрытие устраняет возможность подстановки, эта цена часто невелика, если учесть получение более естественных и понятных моделей.

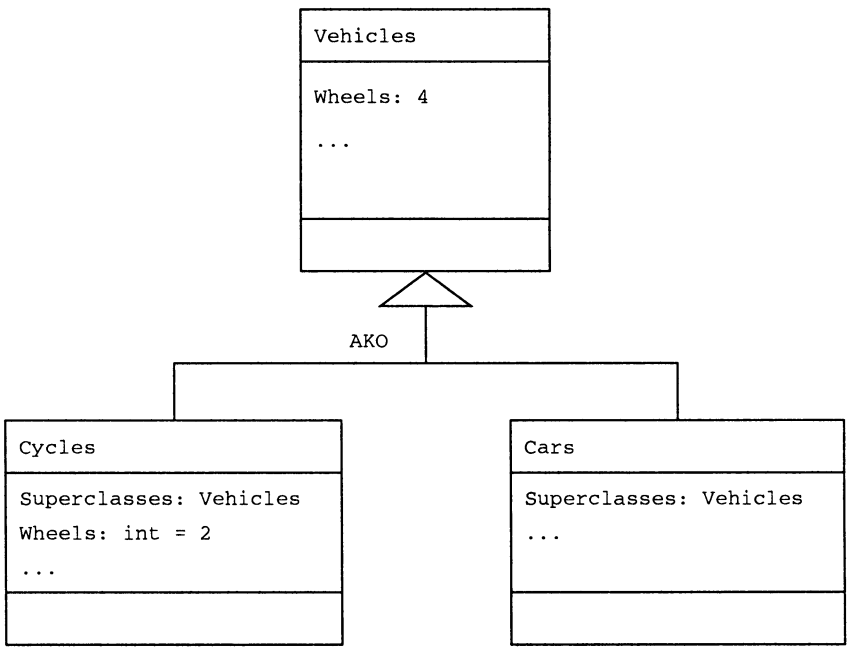


Рис. 1.9. Классы велосипедов Cycles и автомобилей Cars являются специализацией класса транспортных средств Vehicles (отношение AKO). В классе Cycles (велосипеды) перекрывается используемое по умолчанию значение параметра Wheels (количество колес). Классы Cars и Cycles и, возможно, другие классы являются подклассами Vehicles

Если идеи абстракции, рассмотренные в предыдущем разделе, исходят из работ по теории и применению языков программирования, то идеи наследования берут начало в изучении искусственного интеллекта (ИИ), где были разработаны семантические сети [645] и фреймы [562] — методики представления знаний о стереотипных понятиях и объектах. Зависимость между базовыми и более специализированными понятиями в таких сетях реализуется посредством наследования свойств и процедур. Концепции наследования в настоящее время используются в моделировании данных, где широко применяется понятие подтипов логического объекта. В ИИ возможно все: фреймы дают возможность нарушать инкапсуляцию, допускается

множественная и динамическая классификация, а параметры по умолчанию могут передаваться вглубь по структуре наследования. В книге [60] приведены доводы в пользу того, что основанные на фреймовом представлении системы гораздо больше подходят для повторного использования, чем традиционные объектно-ориентированные. Обычно в процессе объектно-ориентированного программирования объекты наследуют методы и свойства от классов, расположенных выше по иерархии, но не наследуют значения атрибутов. Экземпляр наследует только способность иметь определенный тип значений. С другой стороны, в ИИ и в некоторых приложениях баз данных поддерживается возможность наследовать значения. Проиллюстрируем это на примере, к которому иногда будем возвращаться на протяжении всей книги.

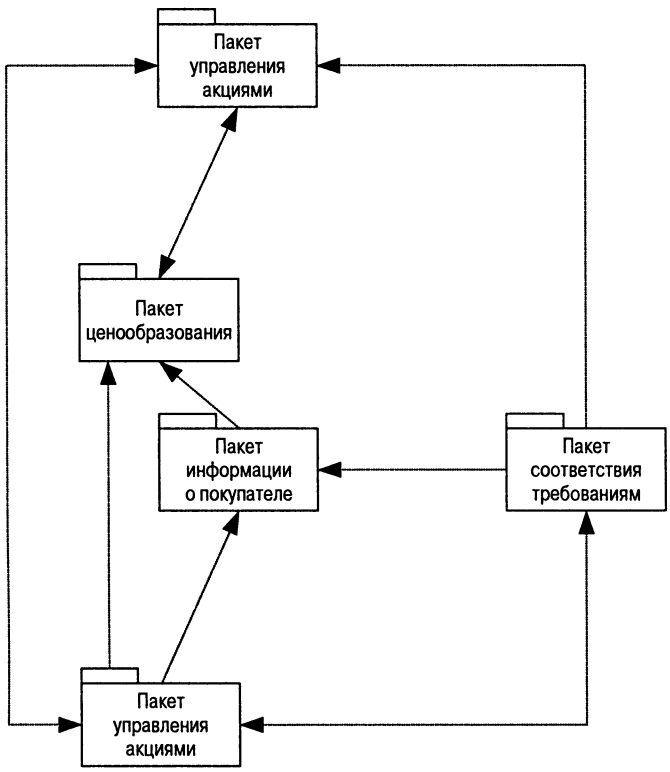


Рис. 1.10. Высокоуровневая структура SACIS

Aardvark в часы досуга

Aardvark Leisure Product — это компания, которая распространяет игрушки и товары досуга для широкой публики. Для этой компании создается информационная система складского учета SACIS (рис. 1.10). Компания Aardvark желает быть первой не только в телефонном справочнике, поэтому руководство решает принять объектно-ориентированный подход к разработке SACIS. Кроме того, система должна быть простой в использовании, чтобы потребители и персонал магазина могли применять ее при распродажах. Это означает, что очень

важную роль играет интерфейс пользователя, и в систему потребуется ввести определенный объем логики, чтобы она могла помочь в поиске товаров, соответствующих нуждам пользователей. Таким образом, SACIS будет экспертной объектно-ориентированной системой баз данных.

Рассмотрим некоторые классы, требуемые в SACIS. В их число по крайней мере должны входить: `People` (люди), `Customers` (потребители), `Adults` (взрослые), `Children` (дети), `StockItems` (товары на складе), `SportItems` (спортивные элементы), `Toys` (игрушки), `Shops` (магазины), `Employees` (служащие) и, возможно, `Suppliers` (поставщики). Классы `Toys` и `SportItems` относятся к классу `StockItems`. Предположим, что существует также класс `Frisbees` и что `F123` принадлежит этому классу. В `StockItems` имеется метод класса, обеспечивающий уведомление об уровне запасов, который наследуется классами `Toys` и `Frisbee`. Если сообщение на уведомление об уровне запасов будет послано `Frisbee`, унаследованный метод проверит значение атрибута `StockLevel` (уровень запасов) и вернет число, представляющее число единиц данного товара на складе. Несомненно, не имеет смысла передавать это сообщение экземпляру `F123`, поскольку отдельные экземпляры класса могут использовать сообщения, направленные всему классу. В этом случае сообщение `delete` (удалить) может быть послано `F123` при продаже этого объекта или лучше (с точки зрения последующей поддержки потребителей) передать сообщение "пометить себя как проданный" с идентификатором покупателя в качестве параметра. Объект `F123` в традиционном объектно-ориентированном программировании наследует все атрибуты `Frisbees`, но не их значения. Аналогично `Frisbees` наследует свойства `Toys`. В системе ИИ мы бы разрешили наследование значений, в частности значений, присваиваемых по умолчанию. Вот почему. Игрушки обладают атрибутом "безопасно для детей", и разумно предположить, что большинство игрушек действительно безопасны. Таким образом, по умолчанию это свойство может принимать значение "да" для всего множества объектов. По мнению автора, удобно наследовать это значение и для класса `Frisbees`, и для экземпляра `F123`, чтобы потребитель мог недвусмысленно запрашивать "это безопасно?" и получать однозначный ответ "да". Для экземпляра `F124`, изготовленного по техническим требованиям заказчика и содержащего цепочки, прикрепленные по периметру, это значение будет перекрываться противоположным. В таких языках, как Smalltalk и C++, все это должно делаться в приложении. В системах ИИ некоторая часть этих действий выполняется автоматически.

Классификационные структуры этого типа реализуют частичный полиморфизм включения, т.е. сообщения, посланные `Toys`, будут поняты `Frisbees` и `F123`, если они не перекрываются.

Композиция и агрегирование

Не все структурные зависимости имеют семантику наследования, однако практические объектно-ориентированные языки должны обеспечивать обработку композиционных структур. В дополнение к наследованию, обобщению и классификации, тут будет рассмотрен другой вид структур: **композиция** (composition), **агрегирование** (aggregation) или структура **APO** (a-part-of). Самый типичный пример — это составной объект, такой как машина, состоящая из корпуса, колес и двигателя. В свою очередь, колесо может состоять из оси, спиц, диска и покрышки.

На рис. 1.11 для обозначения агрегирования используется ромбик. Это обозначение будет более детально объяснено в главе 6. Неоднозначных фраз, подобных "имеет", следует избегать всеми средствами. Аналогичным способом могут быть смоделированы другие ассоциативные структуры, такие как принадлежность, подобие, долг или даже аналогии. В главе 6 будет показано, что эти структуры лучше рассматривать как ассоциации.

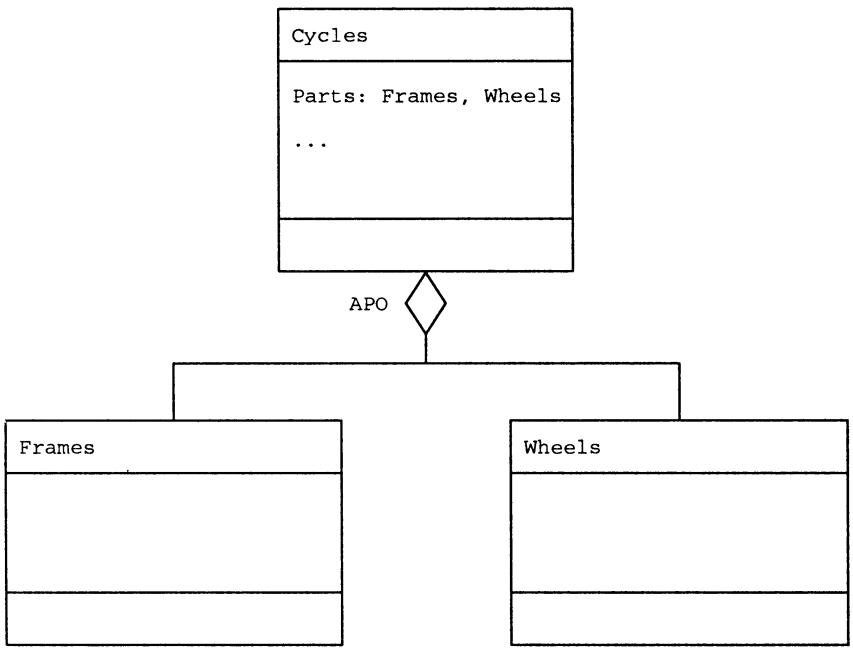


Рис. 1.11. Иерархия композиции, показывающая, что объект `Cycles` состоит из `Frames` (корпуса) и `Wheels` (колес) (и, возможно, из других составляющих). Аббревиатура APO означает A Part Of

Множественное наследование

Возвращаясь к семантике наследования, следует рассмотреть случай, в котором объект или класс может иметь два (или более) родительских объекта. Хорошим примером является рыбка гуппи, которая, как типичная аквариумная рыбка, является экземпляром и класса `Fish` (Рыб), и класса `Pet` (Домашних животных). Гуппи должна наследовать свойства обоих классов. Эта возможность называется множественным наследованием. Проблемой множественного наследования является то, что свойства, унаследованные от двух (или более) родительских классов, часто бывают полностью или частично противоречивыми. Вернемся к нашему примеру: рыба живет в море, а домашнее животное — в доме хозяина, так где должна жить рыбка гуппи? Существует много методов решения этой задачи в каждом конкретном случае. Наиболее привычный метод — это дать возможность системе сообщить о противоречии пользователю и запросить у него значение спорного атрибута. В качестве альтернативы некоторые системы позволяют проектировщику создать процедуру, разрешающую такие противоречия автоматически. Имеются также предложения об объединении ответов в составное

или компромиссное решение (см., например, интерпретацию в приложении А). Другие предложения включают максимальный скептицизм и предположение, что значения являются неизвестными [396].

Возможны два типа конфликтов: имен и значений. В литературе об объектно-ориентированном программировании обычно упоминается только конфликт имен, несмотря на то что в литературе по искусственному интеллекту большее значение придается конфликтам значений, описанным в предыдущем абзаце. Конфликт значений происходит тогда, когда атрибут наследует от порождающих классов два разных значения. Это могут быть значения по умолчанию, если объект сам является классом, или реальные значения экземпляра, если речь идет о наследовании свойств экземпляром (IsA). Конфликт имен происходит тогда, когда два родительских класса содержат разные свойства или методы с одним именем. Конфликты имен обычно возникают в связи с наследованием метода, и в работе [778] перечислены следующие семь стратегий разрешения конфликта, используемые в системе Flavors.

- Вызов наиболее конкретного метода.
- Вызов всех методов в порядке следования или в обратном порядке.
- Выполнение первого метода, возвращающего ненулевое значение.
- Выполнение всех методов и возвращение списка данных.
- Вычисление суммы всех возвращаемых значений.
- Вызов всех демонов `before` (до), а затем вызов всех демонов `after` (после).
- Использование второго аргумента для выбора одного метода или подмножества методов.

К вопросу стратегий разрешения конфликтов мы вернемся в главе 6. Понятие демонов будет объяснено позже по тексту. Можем сказать, демон — это процедура, которая активизируется при изменении данных, а термины `before` и `after` означают, запускается ли процедура в начале изменений или после их завершения; подобно пред- и постусловиям.

Еще один способ избежать некоторых опасностей множественного наследования — отделить наследование интерфейсов от наследования реализации. Этот способ иллюстрируется в языке Java и обсуждается в главе 3.

Роли

И множественное наследование, и отсутствие динамической классификации обнажают вопрос, который до сих пор не обсуждался, — проблема роли. На рис. 1.4 было показано, что `Employees` является классом. Это было бы справедливым в платежной системе, но в целом пребывание в роли сотрудника — это вопрос преходящий. Другими словами, `Employees` означает **роль** (role), а не класс. Поскольку в объектно-ориентированных языках программирования экземпляры не могут изменять свой класс, Анна Арбейтер не может перейти из категории `Students` (студенты) в категорию `Employees` (сотрудники) или из класса `Employees` в `WelfareClaimants` (клиенты департамента социального обеспечения). Чтобы сохранить идентичность Анны, надо сделать ее экземпляром класса `People` и связать с этим классом атрибут экземпляра, отвечающий за сохранение статуса занятости. В качестве альтернативы можно смоделировать роль как класс и использовать шаблоны проектирования `State` (состояние) или `Visitor` (посетитель) (см. главу 7) для перемещения экземпляра из

одного класса в другой. Самая большая опасность в моделировании ролей как классов состоит в том, что это может привести к сверхсложности и созданию огромных структур множественного наследования.

Жизнь без конфликтов в Aardvark

В SACIS необходимость множественного наследования может возникнуть для класса `YoungCustomer` (юный потребитель), который должен наследовать свойства и линию поведения как `Customers`, так и `Children`. Отметим, что некоторые дети могут еще не быть потребителями и что некоторые потребители — взрослые. Класс `YoungCustomer` наследует свойства и методы у класса детей `Children`, которым не продают опасные товары, и это свойство должно перекрывать более слабые значения, унаследованные от класса `Customer`. Это еще раз показывает необходимость стратегий разрешения конфликтов в системах, которые поддерживают множественное наследование. Еще более замысловатая стратегия требуется при продаже таких товаров, как велосипеды, которые являются и игрушками, и спортивным инвентарем. Типичная игрушка — это недорогой товар с коротким временем жизни, не подлежащий техническому обслуживанию. Экземпляры класса `SportItems`, с другой стороны, часто дорого стоят и нуждаются в систематическом профилактическом обслуживании.

Отметим (рис. 1.12), что `CheckValidItem` (проверка соответствия товара) — это разная операция для классов `Customers` и `Children`, потому что первая проверяет легальность товара, а вторая — легальность и безопасность.

Как было показано, в классическом объектно-ориентированном программировании классы наследуют методы и атрибуты (способность иметь значение некоторого типа), а в искусственном интеллекте экземпляры могут наследовать даже значения. В этой книге иногда предпочтение отдается последнему. Это позволяет лучше оперировать семантическими понятиями, такими как значения по умолчанию. Такой подход также выявляет различия между зависимостями AKO и IsA. Наследование классом свойств интерпретируется как отношение AKO (A Kind Of), а наследование экземпляров — как зависимость IsA. Это отличие аналогично различию между включением и принадлежностью в теории множеств. Напомним, что объектно-ориентированное программирование запрещает множественную классификацию. На рис. 1.13 иллюстрируется естественность множественного наследования и множественной классификации.

Наследование обеспечивает расширяемость. В систему можно добавить новую разновидность объекта без необходимости модификации существующего кода. Таким образом, наш гипотетический сотрудник с Ганимеда (Ganymede), спутника Юпитера, может относиться к новому классу `AlienEmployees`, наследующему свойства и методы `Employees` и имеющему дополнительные свойства. Некоторые функции в этом новом классе могут перекрывать функции базового класса. Таким образом, наследование гарантирует, что функции кодируются только однажды.

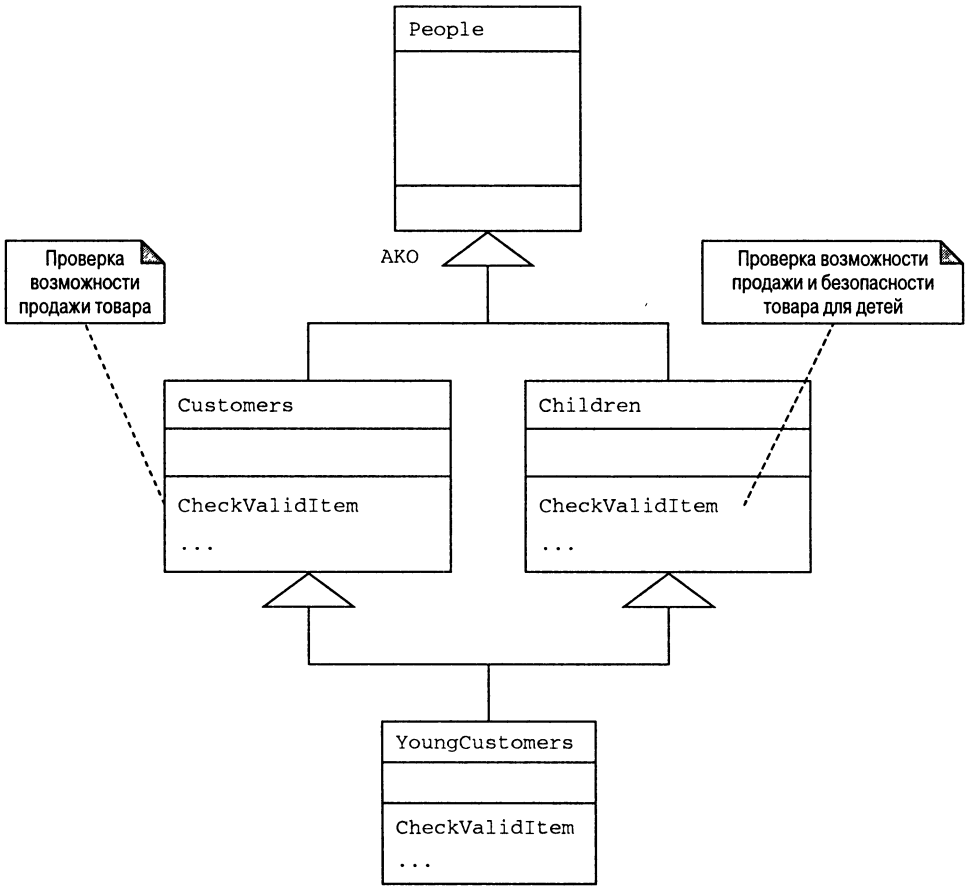


Рис. 1.12. Множественное наследование в SACIS

1.3.3. Инкапсуляция, наследование и объектная ориентация

Умный читатель должен был обратить внимание на то, что в разделе 1.3.1 не было возможности избежать преждевременного упоминания о наследовании. К сожалению, это свидетельствует о том, насколько близки концепции инкапсуляции и наследования. Связь обеспечивается через такие концепции, как полиморфизм, перекрытие, объектная идентичность и передача сообщений.

Наследование часто представляется просто как особый случай полиморфизма, но эта концепция настолько богата и естественна, что данная точка зрения отражает ее не полностью. Пока дело касается языков программирования, эта типично теоретическая точка зрения является обоснованной и полезной. В контексте спецификации и проектирования она более ограничена. Как обсуждалось выше, с этой точкой зрения не совсем согласуется наследование значений.

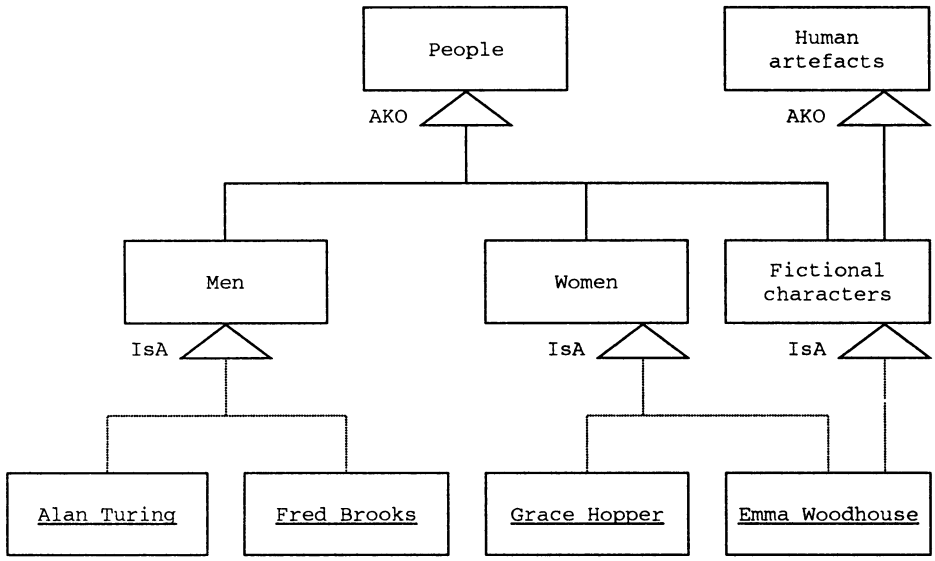


Рис. 1.13. Связи AKO и IsA в структуре классификации

Как будет показано в следующей главе, одно из ключевых преимуществ объектно-ориентированного подхода — возможность повторного использования. Однако с этим требованием необходимо быть предельно осторожным. Безусловно, абстракция обеспечивает возможность повторного использования, но существуют и другие точки зрения [720], [745], согласно которым наследование и делегирование функций могут помешать достижению этой цели. Причина состоит в том, что при наследовании детали реализации иногда открываются клиентам объекта. Кроме того, может быть открыта сама иерархия, так что ее нельзя будет безопасно изменять. Например, если стек определен как частный случай класса `List` (список), то он может унаследовать реализацию операции `head` (голова) как метод `top` и просто исключить несоответствующие операции, такие как `length` (длина). Если реализация стека изменится и станет более эффективной, существует опасность, что клиенты будут зависеть от старой реализации, как частного случая списка. Эта проблема становится еще более существенной, если разрешено множественное наследование. К счастью, Снайдер (Snyder) показал, что аккуратное проектирование позволяет избежать этой проблемы. С другой стороны, наследование частично обеспечивает возможность повторного использования. Без наследования повторное использование многих классов в чистом виде маловероятно.

Применение сложных схем наследования в определенном роде усложняет систему и может свести на нет возможность повторного использования. Компенсирующее преимущество расширяемости — это улучшенная структурная семантика приложения. Чем сложнее система, тем труднее ее поддерживать, тем богаче ее семантика и тем специфичнее сама система. Следовательно, снижается вероятность многократного использования ее компонентов. Однако существуют некоторые приложения, где простое решение — это неудачное решение. Использование сложных схем наследования может быть обоснованным, например, в приложениях из области ИИ и экспертных систем. Именно для таких приложений разрабатываются подобные схемы. По мере того как методология экспертных систем находит свое применение в

традиционных системах, с этим вопросом все чаще сталкиваются аналитики и проектировщики. Например, в системе SACIS задача соответствия товара клиенту требует решения в классе экспертных систем.

Еще несколько определений

Перед тем как завершить раздел основных понятий, нужно добавить несколько определений.

Ключевым понятием объектно-ориентированного подхода является **само-рекурсия** (self-recursion) или ссылка на себя. Это подразумевает, что объекты могут рекурсивно передавать сообщения собственным методам или направлять сообщения себе. В Smalltalk ключевое слово `self` используется для обозначения экземпляра, от имени которого совершается операция, а не класса, который содержит описание этой операции. Это подразумевает, что операция относится к объекту только во время выполнения программы. Еще одна точка зрения на этот вопрос — объект должен знать о собственной уникальной идентичности и должен иметь возможность хранить себя как значение одного из своих собственных атрибутов. Например, в классе сотрудников атрибут `ManagedBy` (руководитель) для экземпляра `J. Smith` может содержать значение `J. Smith`, если J. Smith случайно окажется управляющим директором компании Aardvark. Фактически каждый объект должен содержать ссылку на любой объект, которому он может передать сообщение, если подобная ссылка не передается как параметр.

Таким образом, объектно-ориентированная система программирования, проектирования или анализа должна предоставлять средства инкапсуляции и наследования, а также обеспечивать понятие идентичности объекта и само-идентифицируемости. Для каждого объекта необходимо объявить, какие внутренние данные — аспекты его состояния — можно изменять явно. Его внешнее поведение полностью описывается передачей сообщения.

Теперь можно дать определение **объектно-базированному** программированию как средству разработки программ, поддерживающему инкапсуляцию и идентичность объектов. Иными словами, методы и атрибуты сокрыты внутри и являются закрытыми для объектов, а сами объекты имеют уникальные идентификаторы. Поддержка классов отсутствует или обеспечивается в минимальном объеме, т.е. не поддерживается множественная абстракция. Другими словами, объекты не принадлежат абстрактным классам с отдельной идентичностью. Кроме того, не поддерживается наследование. Ada — это типичный объектно-базированный язык.

Классово-базированные языки, такие как CLU, включают понятие множественной абстракции на уровне экземпляр/класс, но не поддерживают наследование абстрактных классов, которые не могут иметь конкретных экземпляров. Классово-базированные системы обладают всеми свойствами объектно-ориентированных систем; они наследуют их.

Объектно-ориентированные системы обеспечивают наследование свойств и объектно-базированных, и классово-базированных систем, а также дополнительно поддерживают полное наследование между классами, т.е. и экземпляры, и классы наследуют методы и свойства класса (классов), к которому они относятся. Некоторые объектно-ориентированные системы допускают наследование значений атрибутов экземплярами на уровне класса. Объектно-ориентированные системы также обеспечивают саморекурсию.

Компонентно-ориентированные системы — это (тавтология) системы, построенные на использовании компонентов. Компоненты — это выполняемый модуль, который производится, приобретается и развертывается независимо. Это элемент композиции с четко заданным интерфейсом и явной контекстной зависимостью. Такие интерфейсы обычно соответствуют

стандарту языка определения интерфейсов. Компонент играет свою роль в общей композиции и соответствует классу или коллекции нескольких классов.

Итак, мы рассмотрели всю терминологию, необходимую для рассмотрения практических преимуществ и ловушек объектно-ориентированного подхода. Теперь можно приступать к изучению некоторых конкретных языков, чтобы оценить, насколько они соответствуют теоретической идее, описанной выше. Приведенная в главе терминология описана в словаре терминов. Его можно использовать как терминологическую *памятку*.

1.4. Резюме

В этой главе представлена терминология объектно-ориентированных методов и объектно-ориентированного программирования. Обсуждение начато с терминологии в ее историческом и коммерческом контексте, рассмотрены ее истоки, а также связанные с ней ключевые понятия разработки программного обеспечения. К ним относятся следующие.

- Обеспечение возможности повторного использования и расширяемость модулей
- Индустриализация процесса разработки программного обеспечения
- Создание систем с открытыми интерфейсами и совместным использованием ресурсов
- Понимание значения спецификации

Объектно-ориентированный язык (или система) обеспечивает две характерные особенности: инкапсуляцию и наследование.

Абстракция — это процесс выявления существенных объектов в приложении и пренебрежения несущественными свойствами. Абстракция дает возможность повторного использования и сокрытия информации за счет инкапсуляции. Инкапсуляция заключается в сокрытии реализации объектов и открытом провозглашении спецификации их поведения посредством множества атрибутов и операций. Структуры данных и методы, которые обеспечивают выполнение этого принципа, являются закрытыми для объекта.

Тип или класс объекта аналогичен типу данных или типу сущностей с инкапсулированными методами. Данные и методы инкапсулированы и скрыты внутри объектов. Классы могут иметь конкретные экземпляры. Термин *объект* в этой книге означает или класс, или экземпляр.

Наследование — это возможность обобщения и конкретизации понятия или его классификации. Подклассы наследуют свойства и методы от суперклассов и могут добавлять собственные или перекрывать унаследованные. В большинстве объектно-ориентированных языков программирования экземпляры наследуют все без исключения свойства своих классов и только их. Множественное наследование имеет место, когда класс является подклассом более чем одного класса. Наследование обеспечивает расширяемость, но может затруднить возможность повторного использования. Множественное наследование — это мощный инструмент. Однако оно вызывает дополнительные проблемы, и поэтому его нужно использовать с осторожностью.

Объекты взаимодействуют только путем передачи сообщений. Полиморфизм или перегрузка улучшает удобочитаемость текста и производительность программиста, но может привести к снижению эффективности самой программы.

```
объектно-базированный = инкапсуляция + объектная идентичность
классово-базированный = объектно-базированный + множественная
абстракция
объектно-ориентированный = классово-базированный + наследова-
ние + саморекурсия
компонентно-ориентированный = объектно-ориентированный + внеш-
ние интерфейсы
```

1.5. Дополнительная литература

Первыми работами по объектно-ориентированному программированию, возможно, являются [207], [435], [408] и [303].

Существует множество хороших вводных книг по объектно-ориентированному программированию и его преимуществам, хотя в них и не рассказывается об объектно-ориентированных методах вообще. На мой взгляд, наилучшая книга — это [544], основная задача которой — описание языка Eiffel. Однако в этой книге также на высоком уровне представлено введение в основы объектно-ориентированного подхода. Во втором издании книги [549] изложена современная версия тех же идей, а также представлен значительно более завершенный обзор методов из данной научной области. Однако теперь эта книга стала очень объемной и несколько утомительной. Еще одна чудесная книга такого же типа [198] представляет описание языка Objective C. Ее можно назвать хорошим высокоуровневым введением в объектно-ориентированные концепции, кроме того, она весьма хорошо написана. Ни одна книга не содержит достаточно материала об объектно-ориентированном анализе и базах данных, но в [549] изложены интересные идеи, относящиеся к вопросам объектно-ориентированного проектирования. Книга [98] — это самое первое введение в объектно-ориентированное программирование, его концепции и проектирование. Лучшими введениями в объектную технологию на уровне менеджмента проектов являются [750] и [548].

Те читатели, которые желают изучить этот вопрос глубже, могут рассмотреть огромный объем исследовательских статей, часть из которых приведена в списке литературы в конце книги. Две старые, но полезные и доступные работы [712] и [448] содержат некоторое количество подлинно технического и чисто теоретического материала. Работа [516] — это образец совсем недавних исследований этого типа. Книга [83] объединяет несколько хороших вводных статей, а [828] — это огромная коллекция статей по всем аспектам объектно-ориентированных методов. Труды ежегодных конференций OOPSLA и TOOLS всегда включают много полезного и современного материала.

В качестве источников прикладной и теоретической информации следует упомянуть два журнала — это *Journal of Object-Oriented Programming*, JOOP (журнал по объектно-ориентированному программированию) и *Object-Oriented Systems* (Объектно-ориентированные системы), который содержит больше теоретического материала.

Различие между объектно-базированными, классово-базированными и объектно-ориентированными системами изначально было сделано в статье, содержащейся в уже упомянутой ранее книге [712].

В этой книге под объектами понимаются и классы, и экземпляры, несмотря на то что в современных работах объекты и экземпляры различаются. Автор решил не нарушать этой негласной договоренности, что, надеюсь, покажет остальная часть книги.

1.6. Упражнения

1. Выберите **два** характерных свойства ОО подхода.

   а) полиморфизм
   б) наследование
   в) возможность повторного использования
   г) абстракция
   д) инкапсуляция
   е) обобщение
   ж) сокрытие информации
   з) объектная идентичность
   и) динамическое связывание

2. Какая концепция искусственного интеллекта является наиболее близкой к понятию объекта?

   а) слот
   б) механизм вывода
   в) база знаний
   г) фрейм
   д) фацет
   е) правило

3. Какая разница между перечисленными ниже понятиями (сформулируйте по одному предложению для каждого пункта)?

   а) экземпляр и класс
   б) тип данных и класс
   в) класс и роль
   г) тип объекта и тип сущности
   д) класс и компонент
   е) динамическое связывание и полиморфизм
   ж) обобщение и наследование
   з) наследование и классификация

4. Обоснуйте включение атрибутов в описание класса.

5. Укажите различия между чисто ОО стилем наследования и наследованием, применяемым в ИИ-системах. Проследите, поддерживают ли они перечисленные ниже концепции.

   а) динамическая классификация
   б) динамическая специализация
   в) множественное наследование
   г) множественная классификация
   д) наследования значений
   е) перекрытие
   ж) заменяемость

6. Дайте определение идентичности объекта.

7. Сформулируйте определения и приведите примеры следующих понятий.

   а) атрибут/метод класса
   б) атрибут/метод экземпляра

8. Что такое множественное наследование? Когда оно используется?

9. Прокомментируйте следующее высказывание. *"Множественное наследование чрезвычайно опасно; его необходимо запретить. Даже одиночное наследование необходимо устранить из наших языков и методов."*

10. Прокомментируйте следующее высказывание. *"Объектная технология умерла. Ей на смену пришла компонентно-ориентированная разработка."*

11. Постройте диаграммы структур наследования и композиции для классов, упомянутых при обсуждении системы SACIS выше в этой главе.

2

Преимущества объектно-ориентированного программирования и объектно-ориентированных методов

Единственная цель сочинительства заключается в том, чтобы позволить читателям еще больше наслаждаться жизнью или хотя бы лучше переносить ее.

Д-р Джонсон (Критические заметки
о работе Сома Дженинса
Исследование природы и происхождения зла)

В этой главе рассматриваются некоторые преимущества объектно-ориентированных и компонентно-ориентированных методов, а также возможности их использования разработчиками коммерческих систем и пользователями. Кроме того, здесь описываются реально существующие и потенциально возможные ограничения и проблемы, связанные с этой технологией. И наконец, анализируются некоторые опубликованные результаты реальных проектов.

Организации, занимающиеся разработкой современного программного обеспечения (как внутренние подразделения, работающие в сфере информационных технологий, так и компании, предоставляющие консультационные услуги

или поставляющие ПО), выбирают объектную технологию, руководствуясь определенными коммерческими соображениями. Как правило, ими движет желание повысить производительность программистов и их способность к взаимодействию, а также добиться высокого качества. Все это является результатом повторного использования. Но достижение "Святого Грааля" — повторного использования — на практике зачастую вызывает затруднения. Ускоренное продвижение на рынок благодаря быстрой разработке и многократному использованию тоже играет не последнюю роль, и многие компании добились на этом поприще заметных успехов. Более полное оперативное отслеживание требований бизнеса имеет огромное значение в более гибкой и приспособленной к изменяющимся требованиям среде — эта тема будет доминирующей в данной книге. Однако самым главным преимуществом объектной технологии, на мой взгляд, является облегчение тяжкого бремени, связанного с необходимостью сопровождения программного обеспечения.

Провозглашая повторное использование основным преимуществом объектной технологии, не надо забывать, что оно присуще не только объектно-ориентированным методам. На протяжении многих лет библиотеки функций демонстрировали надежное многократное использование в стандартной среде языков COBOL и FORTRAN. Однако хорошо известно, что обычная нисходящая декомпозиция приводит к тому, что модули становятся зависимыми от приложений и малопригодными для многократного использования. Инкапсуляция и наследование позволяют максимально повысить потенциал повторного использования, но они не гарантируют реализации этого преимущества на практике. Успешное решение проблемы требует изменений как в организации, так и в самом процессе разработки. Кроме того, необходимо четкое представление основных принципов и определение последовательности действий. Повторное использование — это не моментальное преимущество, и оно требует затрат.

Исследования, проводимые с начала 1970-х годов, показали, что сопровождение программного обеспечения, вне всякого сомнения, всегда требует от организаций, связанных с информационными технологиями, наибольших затрат. Причем многие эксперты считают, что поддержка "съедает" до 95% от всех затрат, выделенных на обработку данных. Эти же исследования показали, что изменение требований составляет примерно 43% от всех запросов на внесение изменений. Я нахожу это вполне естественным и неизбежным и именно поэтому защищаю быструю разработку приложений: чтобы развитие систем не вызывало затруднений. Уменьшение этих цифр хотя бы на 1% во всемирном масштабе приведет к огромной экономии, измеряемой миллиардами долларов.

Как было отмечено в [748–750], коммерческие устремления в сторону более гибких и надежных компьютерных систем в значительной степени объясняются возрастающими темпами изменения окружающей среды, законам которой подчиняется бизнес. Темпы изменения технологий, общества и условий конкуренции не снижаются. Все более глобальная деятельность компаний усиливает этот процесс, но существование разных валют, норм и положений не должно нарушать обычный ход событий. Ввиду децентрализации и распределенности бизнеса потребность в распределенных компьютерных системах возрастает, и местный персонал наделяется большими полномочиями по принятию решений. Возросшая конкуренция способствует повышению качества продукта, а также созданию продукта, удовлетворяющего требованиям конкретного заказчика. При этом все еще сохраняется экономически выгодное массовое производство.

Работа, выполненная Аланом О'Каллаганом (Alan O'Callaghan) и его студентами в университете Де Монтфорт и компании British Telecom [335], подтверждает это. Авторы полагают, что из-за возрастающей конкуренции необходимы гибкость, более короткие сроки поступления на рынок и более высокая производительность как пользователей, так и разработчиков ПО.

Объектная технология — это единственный хорошо известный подход к выполнению вычислений, который способен решать вопросы и гибкости, и производительности одновременно. Об этом уже упоминалось в работе [327]. О'Каллаган подразделяет движущие силы на коммерческие и технические. Коммерческие стимулы заключаются в том, что надлежащая компонентная архитектура должна обеспечивать конкурентное преимущество; непроизводительные издержки должны быть сокращены посредством выработки совместного представления требований и систем как пользователями, так и разработчиками; и инфраструктура программного обеспечения должна удовлетворять требованиям бизнеса. Заметим, что здесь речь идет не только об объектной технологии, но и о совершенствовании разработки требований к системе и практики быстрой разработки приложений. Технические стимулы находятся в развитии, и к ним можно отнести постоянно растущие вычислительные возможности, совершенствующиеся объектно-ориентированные средства разработки, появляющиеся архитектурные стандарты, например CORBA (Common Object Request Broker Architecture), а также создание Internet, корпоративных сетей и приложений на Java. Оказалось, что принятие этой технологии происходило медленно по следующим причинам. Деловые круги скептически относятся к объектной технологии, поскольку она незаметно сформировалась внутри приложений, которые явно не относились к объектно-ориентированным. Примерами могут послужить протокол IIOP (Internet Inter-ORB Protocol) в Web-броузерах и использование объектно-ориентированных баз данных для поддержки популярных Web-узлов. Как оказалось, большие инвестиции в объектную технологию (ОТ) были сделаны, главным образом, "рископоглощающими" организациями, имеющими достаточные средства на научные исследования и разработку и действующими в областях значительного риска (например, финансов или телекоммуникаций). Кроме того, эти организации были достаточно небольшими и поэтому очень гибкими в ведении бизнеса. Но основным и преобладающим препятствием было существование гигантских работающих систем, нацеленных на решение определенных задач.

2.1. Преимущества

Объектно-ориентированные методы в целом и объектно-ориентированное программирование в частности предлагают проектировщикам и пользователям ряд преимуществ. Эти преимущества во многом имеют такой же характер, как и преимущества структурного программирования и проектирования, но при решении некоторых проблем они затрагивают многие основополагающие принципы структурных методов, отражающие направление мышления.

Хотя полный анализ приводится ниже, однако, заглядывая вперед и учитывая замечания по поводу сокрытия информации и наследования, сделанные в последней главе, сейчас можно выделить главные преимущества объектных методов.

- Сопровождение программного обеспечения осуществляется в конкретном месте и поэтому требует меньших затрат и в меньшей степени подвержено ошибкам даже в случае изменения требований при условии, что структура наследования не должна меняться.
- Объектная технология предоставляет компромиссное решение проблем качества и производительности. Хорошо спроектированные объектно-ориентированные системы являются основой для других систем, которые, в основном, собираются из многократно используемых компонентов, что и приводит к повышению производительности.

Многократное использование готовых классов, уже протестированных в процессе разработки предыдущих проектов, позволяет создавать более качественные системы, удовлетворяющие потребностям бизнеса и содержащие меньше ошибок. Возможно, это и есть самое очевидное преимущество объектной технологии.

- Объектно-ориентированное программирование вообще и наследование в частности позволяют строго определять и использовать модули, которые с точки зрения функциональности не являются завершенными и поэтому могут расширяться, не нарушая работы других модулей и их клиентов. При этом система становится более гибкой, ее расширение упрощается, а стоимость поддержки снижается.
- Объектно-ориентированные средства являются инструментами управления сложностью, позволяющими повысить масштабируемость. Разбиение систем на отдельные объекты способствует решению проблемы их расширения. Поэтому нет причин для экспоненциального роста объема работ при увеличении размеров проекта и его сложности, что характерно для обычных систем.
- Распределение работы над проектом тоже происходит естественным образом, что существенно упрощает решение этой задачи. Если на стадии анализа и проектирования в предметной области выделяются объекты, являющиеся прототипами программных объектов, это зачастую оказывается более наглядным, чем нисходящая функциональная декомпозиция.
- Создание прототипов и постоянная эволюция системы значительно упрощают ее сопровождение, поскольку при этом сокращается время ее поступления на рынок и требования искажаются в меньшей степени.
- Модель передачи сообщений упрощает описание интерфейсов между модулями и внешними (или взаимодействующими) системами.
- Переход от концептуального моделирования, через анализ и проектирование, к кодированию почти незаметен. Объекты могут использоваться на всех стадиях моделирования, поэтому велика вероятность того, что программные объекты будут соответствовать некоторым элементам из пользовательского словаря. Это в значительной степени облегчает взаимопонимание между разработчиками и их клиентами.
- Объектно-ориентированные системы потенциально способны гораздо в большей степени выразить смысл приложения и его семантику. Поскольку объектно-ориентированные методы реализованы во многих системах моделирования, они могут быть использованы для моделирования бизнес-сценариев и их изменения. Это свойство делает конечный продукт более "обратимым" и позволяет выполнять обратное проектирование и соотносить программные элементы с требованиями.
- Сокрытие информации посредством инкапсуляции позволяет создавать безопасные системы.
- Методы формального описания в некоторой степени можно сочетать с объектно-ориентированным проектированием. Более подробно этот вопрос рассматривается в главе 6.

- Некоторые приложения нельзя разрабатывать с применением других подходов, и объектная технология, как оказалось, предоставляет единственный способ создания эффективных приложений такого рода. Примерами могут служить графические пользовательские интерфейсы, распределенные системы, системы на основе агентов и системы управления технологическими процессами.

Кроме всего прочего, разработка программного обеспечения связана с изготовлением высококачественных систем при умеренных затратах на выполнение работ, а следовательно, и умеренной стоимости. Попытки решить проблему качества программного обеспечения берут свое начало от новаторских идей в языках программирования и от различных структурных подходов к разработке систем. Если вы услышали, что до 80% стоимости системы составляет стоимость программного обеспечения и что даже квалифицированные программисты в принципе не способны создавать работающий без ошибок код, то, значит, в ходе выполнения проекта были сделаны серьезные упущения. Использование структурных методов, языков программирования четвертого поколения, CASE-средств, технологий прототипирования, систем баз данных и генераторов кода — все это попытки решения данной проблемы. Возможности применения структурных методов рассматривались в отчете Butler Cox в 1990 году. В этой работе было показано, что при работе над проектом на основе структурного метода действительно снижалась производительность специалистов и качество конечного продукта. Более поздняя публикация компании Butler Cox (в настоящее время CSC Research) подтвердила и уточнила эти данные. Согласно утверждениям некоторых авторитетных источников, при разработке действительно сложных приложений использование языков четвертого поколения также может послужить причиной снижения производительности, главным образом из-за ограниченности языков очень высокого уровня и необходимости перехода на кодирование с использованием языков третьего поколения и обратно.

Толчком к использованию структурных методов во многом послужило то, что многие системы либо были не завершены, либо не использовались, либо в них содержались существенные ошибки. На рис. 2.1 представлена "судьба" некоторых оборонных проектов США 1970-х годов. Заметим, что эти системы, в основном, были предназначены для больших ЭВМ и написаны на языках типа COBOL. Возможно, именно поэтому сравнивать их с системами, разработанными с помощью современных средств, не правомерно. Однако суть от этого не меняется: что-то не в порядке. Более поздние отчеты Standish Group (1995–1997 гг.) только подтвердили эту информацию. Согласно их оценке, почти 59% проектов США были аннулированы или выходили за рамки выделенных на них ассигнований.

Авторы работы [487] пошли иным путем, они исследовали примерно 500 крупных проектов и проанализировали причины запросов на сопровождение (рис. 2.2). Дело в том, что большая часть запросов на изменение, если они не вызваны ошибками в спецификации, неизбежна. Конечно, фраза "изменения требований пользователей" иногда может означать неудовлетворительную спецификацию, но она отражает также и динамический характер современного бизнеса. В постоянно меняющемся мире программные системы должны быть легко приспосабливаемыми, и это, по сути, полностью соответствует способности объектно-ориентированных систем быть наращиваемыми и модульными. Среди типов изменений можно также выделить изменение структур данных, и здесь инкапсуляция является наиболее перспективной технологией, придающей системам устойчивость к изменению реализации. Например, когда корпорация British Telecom изменяет формат лондонских телефонных номеров (что они периодически и делают), то программное обеспечение, написанное на основе объектно-ориентированных методов, меньше страдает от изменений подобного

рода, поскольку внутреннее представление номера телефона является скрытым и изменения затрагивают только этот объект; остальные же части системы в кардинальных изменениях не нуждаются.

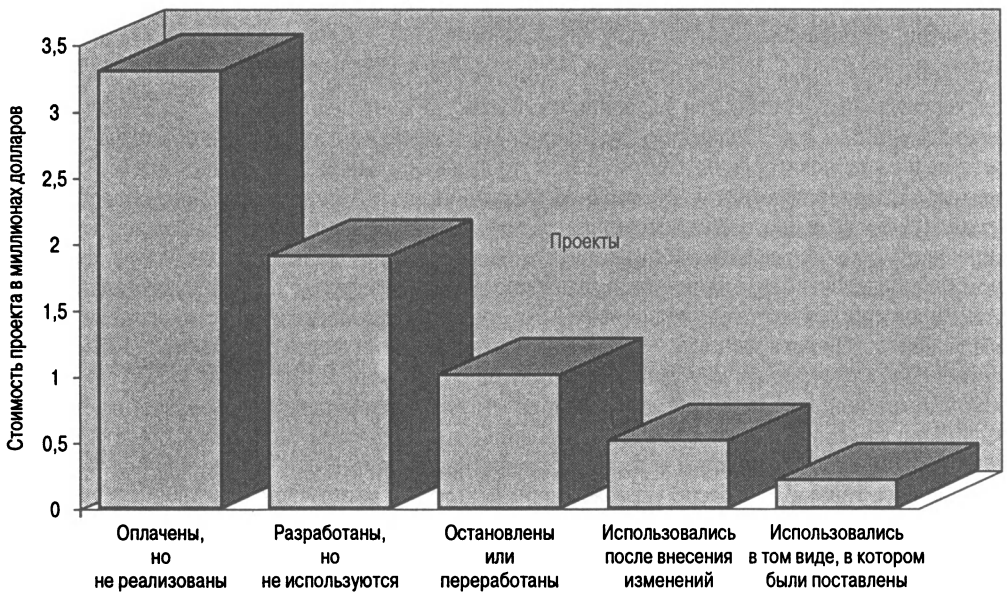


Рис. 2.1. "Судьба" оборонных проектов США согласно статистическим данным правительства США (взято из [182])

Аналогично проблемы, связанные с ошибкой 2000 года, были бы устранены, если бы все программы использовали объекты даты: реализация полей года, предусматривающая 4 цифры вместо 2, была бы для клиентских программ абсолютно незаметной. Конечно, для подобных преимуществ всегда существуют ограничения. Предположим, что ширина бумаги для печатающего устройства оказалась недостаточной для размещения нового программного кода и что печатающее устройство физически не способно использовать более мелкие шрифты.

Нужно учитывать также необходимость выполнения специальных и аварийных изменений. Непредвиденные ситуации в обычном коде зачастую приводят к неожиданным последствиям. В объектно-ориентированных системах это по-прежнему возможно, однако я считаю, что это происходит реже и отслеживается намного лучше, поскольку изменения либо инкапсулированы внутри объекта, затрагивая лишь его внутреннюю реализацию, и могут быть обнаружены только через интерфейс, либо приводят к другой маршрутизации сообщений.

Я полагаю, что прототипы способствуют получению корректных спецификаций. Более подробно этот вопрос рассматривается в главе 9, посвященной вопросам управления процессом разработки систем. При условии правильного управления прототипы могут применяться в процессе традиционной разработки программного обеспечения, а при их комбинировании с объектно-ориентированными методами можно получить дополнительные преимущества. Объектно-ориентированные языки программирования типа Smalltalk являются хорошими языками создания прототипов, а существование многократно используемых модулей на языке

Java способствует не только более быстрому созданию прототипов, но и более быстрому завершению разработки системы. Если прототипирование используется в качестве инструмента описания, то его можно с таким же успехом применять для разработки объектно-ориентированных проектных решений и спецификаций, а не только в языках программирования.

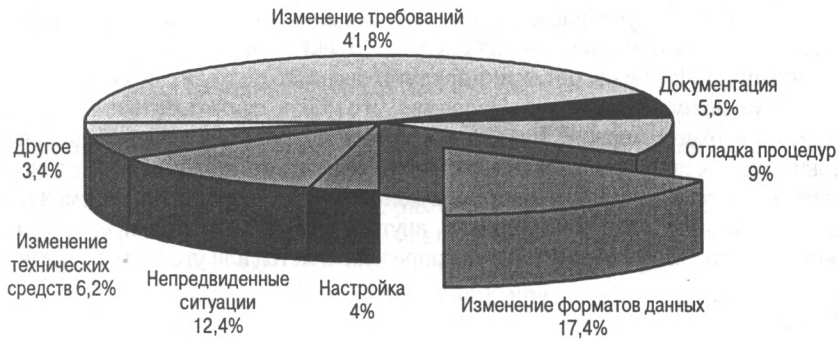


Рис. 2.2. Анализ распределения расходов, связанных с сопровождением программного обеспечения

По моему мнению, высказанная выше мысль по поводу непредвиденных ситуаций указывает на основную проблему, связанную с генераторами кода. Основная идея использования CASE-средств для генерации исходного кода и создания удовлетворяющей требованиям и совместимой в рамках всех проектов документации заключается в том, что все изменения должны вноситься в систематизированном порядке. Если дежурный программист, работающий в ночную смену, столкнется с аварийной ситуацией, то зачастую единственной возможностью ее устранения будет выявление и исправление исходного кода напрямую. У него просто не будет времени на модификацию моделей и документации. Поэтому неизбежно со временем спецификация "отдалится" от исходного текста, и выполнять изменения станет все сложнее. Обычно приверженцы генераторов исходного кода отвечают, что эта проблема вполне поддается управлению и что подобные изменения должны быть запрещены. Однако реальная жизнь диктует свои условия: пользователи не желают, чтобы их системы простаивали на протяжении нескольких часов или дней, пока такие замечательные процедуры не будут выполнены. При решении этой проблемы объектно-ориентированные системы обеспечивают более простое отслеживание непредвиденных ситуаций, позволяя позже с помощью обратного проектирования внести изменения и в спецификацию. При упоминании CASE-средств заслуживает внимания еще одна проблема, связанная с трудностями общего характера, с которыми сталкиваются руководители очень больших проектов. При генерации кода с использованием CASE-средств предполагается, что основные усилия затрачиваются на анализ и архитектурное проектирование. При этом в плане проекта остается совсем немного времени на исправление ошибок, обнаруженных во время тестирования кода. Большинство менеджеров проекта не готовы осознать тот факт, что в самом конце проекта придется перенести срок сдачи системы, чтобы иметь возможность устранить потенциально огромные проблемы. Автору приходилось сталкиваться с некоторыми проектами, при реализации которых CASE-средства не использовались именно из-за описанной выше опасности. Конечно, объектно-ориентированные CASE-средства, которые позволяют генерировать код,

тоже подвержены подобным проблемам. В то же время модульность, реализованная с использованием инкапсуляции, зачастую позволяет тестировать модули независимо друг от друга и на ранней стадии разработки. Тем самым можно правильно выполнить оценку ошибок, которые придется исправлять впоследствии.

На протяжении значительного времени нисходящая декомпозиция считалась "лекарством" от многих "болезней", хотя истинные ее сторонники на самом деле никогда не рассматривали ее в качестве механизма управления проектом, а защищали как метод *описания* систем. В этом смысле она действительно очень полезна. Объектно-ориентированные методы не являются нисходящими. В рамках объектно-ориентированного подхода считается, что реальные системы зачастую не имеют вершины. Например, что можно считать функцией операционной системы самого верхнего уровня? Разделение работы сопровождается размещением объектов в различных пакетах, а не разбиением задач на подзадачи. Такой вид деятельности почти всегда зависит от специфики приложения и в конце концов приводит к невозможности многократного использования модулей. Конечно, внутри объекта можно по-прежнему использовать функциональную декомпозицию, чтобы определить метод или уточнить прецедент.

Качество

Согласно утверждениям Мейера [544, 548], Соммервиля [723] и некоторых других специалистов, качество программной системы следует оценивать по следующим критериям.

- *Корректность*. Программы должны полностью соответствовать своим спецификациям.
- *Устойчивость и надежность (робастность)*. Программы должны надежно работать даже в аварийных ситуациях.
- *Возможность сопровождения и расширения*. Программы должны легко развиваться и расширяться при изменении требований.
- *Возможность повторного использования и универсальность*. Программы должны строиться из модулей повторного использования.
- *Способность к взаимодействию*. Программы должны легко совмещаться с другими системами. Они должны быть "открытыми системами".
- *Эффективность*.
- *Переносимость*. Программы должны быть переносимы на другие аппаратные средства и операционные системы.
- *Возможность верификации*.
- *Безопасность*. Может потребоваться, чтобы данные, знания и даже функции были эффективно скрыты.
- *Целостность*. Системы нужно защищать от противоречивых изменений.
- *Дружественность*. Для большинства пользователей системы должны быть просты в использовании и предоставлять лишь необходимую информацию.
- *Возможность описания*. Должна существовать возможность создания и поддержки документации.
- *Ясность*.

Объектно-ориентированный подход во многом способствует достижению этих целей, хотя некоторые объектно-ориентированные языки могут проигрывать в эффективности, ясности, робастности или переносимости. В частности, преимуществами объектно-ориентированных систем зачастую считается свойственная объектам возможность повторного использования, расширяемость объектно-ориентированных систем и тот факт, что формальные методы описания могут хорошо сочетаться с объектно-ориентированным подходом.

На протяжении длительного времени решение проблемы создания повторно используемого кода было целью проектировщиков систем. Некоторые успехи в этой области были получены при разработке повторно используемых функций, размещенных в библиотеках, которые предназначались для хранения часто используемых математических и статистических функций. Однако на практике даже в этом случае зачастую возникали трудности, поскольку пользователям приходилось запоминать огромное количество информации, связанной с длинными списками входных параметров. Это приходилось делать, поскольку данные тесно связаны с областью применения программного пакета. Причина возникновения такой ситуации отчасти связана с популярностью использования декомпозиции "сверху внизу", когда задача разбивается на составные части (достаточно небольшие, чтобы их можно было решать отдельно и за короткий срок). Эти части в значительной степени зависят от самого способа декомпозиции. Функциональная декомпозиция "сверху вниз" по своей природе зависит от специфики приложения. В результате полученные модули зачастую не являлись универсальными, и программисты заново изобретали одни и те же решения в различных контекстах.

Некоторые специалисты утверждают, что при широком применении механизма повторного использования затраты на разработку могут уменьшиться почти на 20% и что даже небольшая экономия в 1% даст огромное преимущество перед конкурентами. По другим оценкам 70% усилий программистов направлено на поддержку существующего кода, а 80% этих усилий затрачивается не на совершенствование, а на обеспечение корректности. Специалисты Министерства обороны США считают, что затраты на исправления в десять раз превышают расходы на новые разработки. Таким образом, в области разработки программного обеспечения первоочередное внимание должно уделяться всему, что способствует улучшению спецификаций и повторному использованию.

Объектно-ориентированное проектирование в значительной мере способствует решению проблемы повторного использования, возможности сопровождения и обеспечения способности к взаимодействию, поскольку при этом используется три основных приема: проектирование "снизу вверх", инкапсуляция и наследование.

Проектирование "снизу вверх" значительно повышает возможность повторного использования модулей. Для большинства программистов наглядным и простым примером может послужить модуль обработки даты. Если заранее известно, что некоторая система должна использовать даты или производить вычисления с их использованием, то может оказаться очень полезным отступить от существующих системных требований и определить, насколько выгоднее было бы реализовать общий набор процедур манипулирования датой, который можно было бы использовать во всех последующих системах. Таким образом, если в существующей системе дату Пасхи вычислять не требуется, то необходимо реализовать перехват этой функции в пользовательском интерфейсе или даже во всей программе. Функциональная декомпозиция не позволяет идентифицировать подобные требования, и в результате может быть создана процедура, в которой вычисления национальных праздников предполагают кардинальные изменения. Именно инкапсуляция, или сокрытие информации, дает возможность проектирования "снизу вверх".

Еще одним приемом является сама инкапсуляция. Модули, реализованные с применением объектно-ориентированного подхода, доступны исключительно через свой интерфейс. Программисты должны быть знакомы лишь с этим интерфейсом, который эквивалентен описанию функции модуля. Реализация модуля скрыта и никак не связана с его использованием в системе. Сокрытие информации способствует также повышению безопасности. Простой набор интерфейсов (т.е. все они имеют небольшой размер, заданы явным образом и немногочисленны) повышает способность к взаимодействию и степень дружественности. Как упоминалось в главе 1, одним из самых ранних применений объектно-ориентированного программирования было проектирование пользовательского интерфейса.

Наследование, обобщение и другие виды полиморфизма упрощают обработку исключений и повышают возможность расширения систем. В системах с наследованием функции могут добавляться посредством добавления дочерних объектов, которые наследуют описание и реализацию своих родителей, но все же ведут себя иначе в тех областях, которые к ним относятся. Таким образом, последовательные изменения становятся понятнее и проще реализуются. В последней главе отмечено, что при некорректном использовании наследования можно нарушить принцип абстракции и тем самым чрезмерно усложнить систему. Однако, с другой стороны, понятие расширяемости тесно связано с преимуществами повторного использования. Если модуль нельзя расширить, то его можно будет повторно использовать далеко не во всех приложениях.

Понятия корректности и робастности имеют отношение к методам создания прототипов и формальных спецификаций (см. главы 6 и 7), которые хорошо сочетаются с объектно-ориентированными методами. Если (!) модули правильно сконструированы с учетом их повторного использования в дальнейшем, то вполне вероятно, что они окажутся и более робастными. Каждый раз при использовании они подвергаются дополнительному тестированию.

Возможность сопровождения, повторного использования и верификации обеспечивается одновременно "открытой" и "замкнутой" природой объектно-ориентированных систем. В соответствии с принципом "открытости-замкнутости" Мейера (Meyer), повторно используемые системы должны быть как открытыми (в смысле простоты их расширения), так и замкнутыми (в смысле их готовности к использованию). Пример кода, не удовлетворяющего этому принципу, можно найти практически в любой системе, в которой используются операторы перехода `goto` и `case` обычного языка программирования. Например, рассмотрим следующий фрагмент кода, демонстрирующий обработку ошибок на языке программирования типа C.

```
switch (n)
{case 1:  error = 49;
              print("Предупреждение — Ошибка 49"); continue;
 case 2:  error = 71;
              print ("Выполнение прервано из-за

Ошибки 71"); break;
 case 3:
              break;
}
```

Приведенный фрагмент кода абсолютно непригоден для повторного использования и расширения. Более того, для него очень трудно было бы создать документацию, более краткую, чем сам код. Чтобы добавить новый оператор `case`, т.е. обработку еще одной ошибки,

программист должен был бы взять модуль независимо от самой системы, внести в него требуемые изменения, а затем еще раз откомпилировать. Таким образом, если система не является закрытой, то ее нельзя использовать в процессе внесения изменений. Кроме того, нет никакой гарантии, что на использование существующих функций не повлияют ошибки программиста. Система не является открытой, поскольку для добавления новых функций (ее расширения) потребуется изменить существующий код на уровне реализации. Системы с использованием наследования, в том числе и объектно-ориентированные, действительно подчиняются принципу открытости-замкнутости (open-closed). Кроме того, существуют также такие объектно-ориентированные языки, которые допускают изменение модулей во время работы системы.

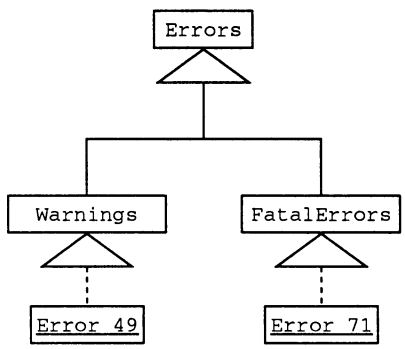


Рис. 2.3. Перехват ошибок с использованием объектно-ориентированного подхода

Основным приемом объектно-ориентированного программирования, предоставляющим возможность расширения, является наследование. Если проблему перехвата ошибок рассмотреть с точки зрения объектно-ориентированного подхода, то станет очевидно, что ошибки можно разбить на несколько типов (рис. 2.3). В рассмотренном простом примере ошибки подразделяются на неисправимые и исправимые (предупреждения). Оба типа наследуют атрибуты и методы всех ошибок в целом, а отдельные ошибки можно рассматривать как экземпляры соответствующего подкласса. Чтобы добавить обработку новой ошибки (или, что еще более важно, нового типа ошибки), достаточно создать новый класс, описать его и создать новый объект этого класса. Так, при обнаружении нового типа ошибки, который нельзя отнести к представленным двум типам, необходимо создать новый подкласс ошибок и описать присущие ему отличительные особенности. При таком подходе способы обработки ранее существующих ошибок не затрагиваются, и при этом необходима компиляция только нового класса.

Конечно, выше приведен тривиальный и нереалистичный пример, однако он демонстрирует ключевую идею: с использованием наследования можно конструировать расширяемые системы. Более реальный пример представлен на рис. 2.4. На основе приведенной диаграммы разработчик может разработать некоторый код для общих функций всех банковских счетов, таких как методы с операторами печати, а специальные функции депозитных и текущих счетов (для США, чековых) разместить в подклассах. Например, для депозитных счетов должен выполняться расчет процентов. При рассмотрении текущего счета, приносящего проценты, можно без проблем добавить новый класс, который будет подклассом либо депозитных

счетов, либо текущих. Кроме того, для наследования функций от счетов обоих типов можно воспользоваться множественным наследованием, которое на рис. 2.4 представлено в виде серой стрелки.

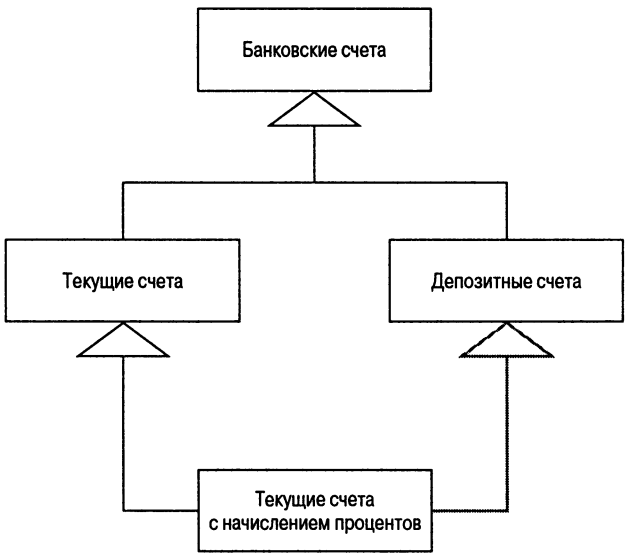


Рис. 2.4. Классификация банковских счетов

И объектно-ориентированное, и функциональное программирование позволяют программисту создавать открыто-замкнутые системы. В данной книге функциональное программирование не рассматривается, однако в следующей главе все же будут затронуты некоторые из связанных с ним вопросов.

Такой внутренне противоречивый принцип открытости-замкнутости играет важную роль при сопровождении программного обеспечения, управлении проектом и особенно при его кардинальном изменении. Именно этот принцип и обусловливает появление методов экстремального программирования (eXtreme Programming) Бека [63]. Системы должны быть открытыми, поскольку в процессе разработки модуля конструкторы редко предвидят все варианты или результаты его применения. Позднее может понадобиться добавить новые функции. Однако если системы не являются замкнутыми и в таком виде поставляются потребителям, то они оказываются абсолютно непригодными. В самом начале разработки открыто-замкнутых систем предполагается, что существуют вполне определенные процедуры управления изменениями. С учетом этого требования и должен рассматриваться эволюционный процесс разработки и объектно-ориентированное программирование. В оставшейся части книги этому вопросу будет уделяться достаточно большое внимание.

Количество различных способов выполнения функциональной декомпозиции определяется законами комбинаторики. Поэтому конкретный способ, скорее всего, будет очень сильно зависеть от специфики приложения. Если же воспользоваться принципом разбиения "снизу вверх" и объектной парадигмой, то можно существенно расширить возможности повторного использования. Однако такой подход является далеко не самым простым. В те времена, когда

поставщики баз данных не были столь предусмотрительны, чтобы определить для даты встроенный простой тип данных, приходилось создавать собственные процедуры их обработки. Автор тоже разрабатывал одну из подобных процедур и оказался более дальновидным, поскольку включил в ее состав алгоритм вычисления дня Пасхи. Описание этого алгоритма содержится в церковных книгах. В то время он позволял вычислить дату всех национальных праздников Великобритании. Оглядываясь назад, можно сказать, что основная проблема заключалась в том, что в разработанную процедуру не входили алгоритмы вычисления дня Рамадана, искупления грехов и Китайского Нового года. Из сказанного можно сделать вывод, что программирование "снизу вверх" с учетом повторного использования является очень дорогостоящим и полностью повторно используемый код будет бесконечно дорогим. Решение состоит в комбинировании обоих подходов программирования: и "снизу вверх", и "сверху вниз". А кроме того, нужно создавать только те функции, которые с большой вероятностью пригодятся для повторного использования. Идея подключаемых контуров, впервые сформулированная в рамках метода Catalysis [204], призвана решить именно эту проблему. В соответствии с этим методом должны быть выявлены точки потенциальных вариаций, которые обрабатываются с помощью вызовов отдельных процедур. Такой канонический подход к программированию "снизу вверх" очень трудно реализовать на практике, поскольку сложно оценить и спрогнозировать затраты на обращение к объектам. Расширяемость объектно-ориентированных систем гарантирует, что непредусмотренные функции можно добавить позднее, однако это повлечет дополнительные затраты. Эта идея более подробно рассматривается в главе 7. Объектно-ориентированное программирование позволяет повысить производительность программистов. Но вместе с тем существует опасность того, что программисты начнут вводить ненужные функции.

В работе [197] приведена концепция так называемых программных интегральных схем (Software Integrated Circuits), в рамках которой программы сравниваются с устройствами, которые в большинстве случаев буквально собираются из стандартных компонентов, как, например, автомобили, радиоприемники и холодильники. В настоящее время эта же идея известна как компонентно-ориентированная разработка (component-based development — CBD). Экономические преимущества использования такого подхода хорошо известны. Почему же для программных систем эта цель выглядит труднодостижимой? Во-первых, это именно тот случай, когда нестандартные программные системы (а к ним относится большинство систем), в отличие от многих промышленных товаров, не производятся в массовых количествах. Другими словами, существует только один конечный продукт. Таким образом, выгоды от увеличения масштабов производства, связанные с массовым изготовлением, в данном случае недоступны. И зачастую финансисты проектов не хотят инвестировать дополнительные работы по разработке чего-либо, что не сразу оправдает затраты при решении ближайших задач. Во-вторых, широкое распространение в недавнем прошлом методов декомпозиции "сверху вниз" привело к разделению видов деятельности, которое сильно зависит от специфичной для приложения декомпозиции проблемы. Это означает, что способ определения границ модулей зависит от декомпозиции, которая, в свою очередь, определяется функциональными характеристиками конкретного приложения, а не естественным делением реального мира на объекты. Объекты могут естественно быть связаны друг с другом во многих приложениях и поэтому могут повторно использоваться. Объектно-ориентированные методы соответствуют именно такому представлению и препятствуют чрезмерному увлечению философией нисходящего разбиения. Возвращаясь к первому утверждению, можно сказать, что руководителей проектов и спонсоров необходимо было бы несколько перевоспитать. Планирование проекта и оценка его стоимости должны производиться на базе не только существующих, но и будущих

проектов. Проблема состоит в том, что в обществе, которое желает как можно быстрее получить прибыль и постоянно сокращает сроки выполнения проектов, почти всегда легче оправдать создание индивидуальных проектов. К сожалению, вполне вероятно, что ввиду такой экономической реальности повторно используемые модули будут создаваться, скорее, по воле случая, а не разумного решения. И технологии, которые волей-неволей способствуют повторному использованию, будут единственными используемыми технологиями. Объектно-ориентированным методам, описанным в этой книге, присуща именно эта особенность. Другими словами, возможность повторного использования проще реализовать при помощи объектно-ориентированных методов. При этом лишь немного повысятся (или останутся на прежнем уровне) суммарные затраты, хотя стоимость первоначальной разработки повторно используемого модуля может оказаться больше, и даже гораздо больше стоимости модуля, специфичного для приложения. По самым смелым предположениям эта стоимость может повышаться в десятки раз. Однако всегда есть небольшая вероятность того, что объектно-ориентированный подход позволит создавать повторно используемые модули без дополнительных затрат. Это возможно в тех случаях, когда объекты выбраны очень удачно, т.е. они соответствуют реальным объектам из предметной области приложения.

Модульность

Важность модульности подчеркивалась во многих теоретических работах, посвященных программному обеспечению. В [617] выдвинута идея использования модулей с целью сокрытия информации. В работе [544] сформулировано пять критериев и пять принципов модульности. Сейчас рассмотрим пять критериев модульности, сформулированных Мейером (Meyer).

1. Разделимость (decomposability)
2. Сочетаемость (composability)
3. Понятность (understandability)
4. Непрерывность (continuity)
5. Защищенность (protection)

Понятие *разделимости* связано с проектированием программного обеспечения и управлением проектом и означает, что системы должны быть разложимы на управляемые фрагменты, упрощающие процесс внесения изменений и позволяющие распределить взаимосвязанные задачи среди отдельных разработчиков или их групп. Технология нисходящей декомпозиции позволяет достигнуть этой цели на основе использования единой функции самого высокого уровня, однако ее использование теряет смысл, если такая реальная "вершина" отсутствует, например, в операционной системе. В этом случае можно также определить модули, связанные друг с другом очень сложными интерфейсами. Объектно-ориентированная декомпозиция является восходящей и базируется на принципах сокрытия информации и использования простых интерфейсов.

Понятие разделимости сильно связано с децентрализацией. Децентрализация способствует упрощению кода, устраняя скрытые зависимости, которые являются причиной значительных изменений кода при относительно небольших изменениях проектного решения. Если каждому модулю системы требуется знать лишь о собственной реализации, а не о серверных компонентах, то расширение его кода не вызовет никаких затруднений. Таким образом, проектные изменения будут изолированными и не будут автоматически оказывать влияния на остальную часть системы.

Сочетаемость (*компонуемость*) — это возможность произвольного комбинирования модулей даже в тех системах, для которых они не разрабатывались. Это свойство является основополагающим для повторного использования и опять же обеспечивается принципом сокрытия информации. Как уже упоминалось, функциональная декомпозиция позволяет осуществить разбиение, однако она не способна обеспечить компонуемость. Одна из замечательных и уникальных особенностей объектно-ориентированного подхода заключается в том, что этот подход поддерживает как компонуемость, так и разделимость. Компонуемость связана с понятием расширяемости, которая позволяет добавлять функции в систему, не прибегая к радикальным "хирургическим операциям". Компонуемость и расширяемость поддерживаются объектно-ориентированными принципами использования небольших интерфейсов (см. ниже) и особенно, как можно увидеть, наследования. Это предполагает использование одного языка программирования. Однако возникает вопрос, возможна ли компоновка систем из объектов, написанных на разных языках. Этот вопрос будет рассматриваться в главе 4. Безусловно, декомпозиция "сверху вниз" может обеспечить разделимость, однако лишь в редких (весьма специфических) случаях она позволит создать компонуемые модули.

Понятность способствует осмыслению системы в целом посредством предварительного рассмотрения ее частей. Этот принцип справедлив даже в том случае, когда вся система обладает новыми качествами, не свойственными ее составным частям. При правильном использовании такой подход позволяет лучше изучить и поддерживать систему. В этом смысле объектно-ориентированные системы настолько понятны, что нет никакой необходимости подробно отслеживать их поведение при передаче сообщений и определять, какие же действия они выполняют.

Непрерывность системы предполагает, что ее небольшие изменения приведут лишь к незначительным изменениям в ее поведении, а также что небольшие изменения спецификации потребуют внесения изменений лишь в некоторые модули. Последнее утверждение касается непосредственно объектно-ориентированных методов. Стоит еще раз подчеркнуть, что это преимущество обеспечивают именно наследование и простые интерфейсы.

Критерий модульной *защищенности* требует, чтобы исключения и ошибочные ситуации либо обрабатывались внутри модуля, в котором они произошли, либо затрагивали лишь немногие, наиболее тесно связанные с ним модули. Например, выполняемая в системах баз данных проверка достоверности внутри описаний сущностей реализует механизм защиты, обеспечивающий невозможность ввода некорректных данных. Это определяется свойствами модуля, в который они вводятся.

Пять принципов, гарантирующих выполнение критериев модульности, были сформулированы Мейером (Meyer).

1. Лингвистические модульные единицы
2. Небольшое число интерфейсов
3. Интерфейсы небольшого размера
4. Явно определенные интерфейсы
5. Сокрытие информации

Выше эти принципы упоминались при обсуждении пяти критериев Мейера. *Лингвистическая модульная единица* означает, что модули системы должны соответствовать простым типам данных или синтаксическим единицам языка или метода, используемого для описания проблемы. В частности, этот принцип поддерживают абстрактные типы данных, в свою

очередь обеспечивающие разделимость, сочетаемость и защищенность. Например, использование во многих языках программирования примитивов с плавающей точкой способствует их многократному использованию в самых разных программах, а типы, задаваемые пользователем, позволяют распространить это преимущество на другие модули. Благодаря этому принципу многие современные программные продукты оказались очень удачными. Базы данных работают с целыми записями и таблицами, точно так же как с примитивами, а системы финансового моделирования и интерактивные системы аналитической обработки (On-Line Analytic Processing — OLAP) — со строками и столбцами. При этом упрощается разработка некоторых приложений, их объединение и тестирование.

Небольшое количество интерфейсов означает, что модуль должен взаимодействовать лишь с ограниченным числом других модулей. Выше уже упоминалось о трудностях, связанных с отслеживанием передачи сложных сообщений. Принцип небольшого числа интерфейсов способствует решению как проблемы непрерывности, так и защищенности. Топологически этот принцип можно реализовать различными способами. Хорошей аналогией может послужить локальная сеть, в которой каждую рабочую станцию и сервер можно рассматривать как объекты, связывающиеся друг с другом посредством передачи сообщений. Такая сеть может быть реализована либо как централизованная сеть в виде звезды, либо как кольцо, в котором связываться могут только соседние узлы, хотя сообщения могут передаваться через ретранслятор. Конечно, реальные сети функционируют по-другому, однако приведенный пример можно спроецировать на организацию объектов внутри системы. Сеть, в которой каждый узел связан со всеми другими узлами, нарушает этот принцип, а ее поддержка и изменение, несомненно, вызвали бы значительные затруднения.

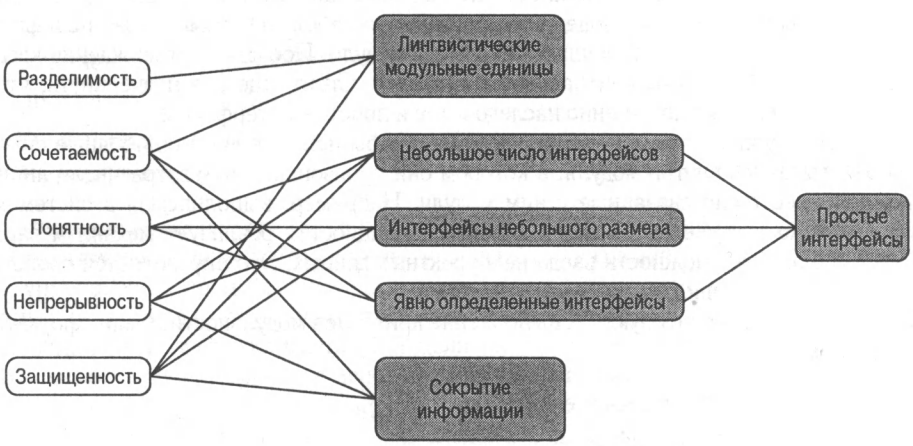


Рис. 2.5. Взаимосвязь пяти критериев и принципов Мейера

В соответствии с принципом *интерфейсов небольшого размера*, с использованием интерфейсов должно передаваться как можно меньше данных. Этот принцип также связан с критериями непрерывности и защищенности.

Интерфейсы должны задаваться явно, поскольку объекты должны использоваться через их описание, а не реализацию. Тогда при компоновке и разбиении связи между модулями станут достаточно очевидными. Непрерывность, также тесно связанную с этими рассуждениями,

можно увидеть в спецификации модулей. Совместное использование данных повышает эффективность их хранения, однако может привести к нарушению этого принципа, если соответствующий механизм явно не определен в интерфейсах взаимодействия модулей.

Последние три принципа могут быть объединены в один принцип *простых интерфейсов*. Вообще, объектно-ориентированные системы должны быть написаны таким образом, чтобы все взаимодействие с объектами выполнялось через их интерфейсы (или спецификации), а не через их реализации. По возможности эти интерфейсы должны быть максимально простыми: небольшими, четко определенными и в малом количестве. Это и есть тот общий принцип, который будет постоянно упоминаться в этой книге.

Принцип использования простых интерфейсов обеспечивает основу того, что критерию компонуемости могут удовлетворять системы, написанные на разных объектно-ориентированных языках. Если же не придерживаться этого принципа, то подобная возможность, скорее всего, будет неиспользованной.

Принцип *сокрытия информации* является наиболее важным. Согласно этому принципу модули могут инкапсулировать проектные решения, принятые при их создании. С точки зрения терминологии объектно-ориентированных методов это означает, что модули используются через свои спецификации, а не реализации. Вся информация о модуле (связанная и с данными, и с функциями) инкапсулирована внутри него и, если модуль специально не определен открытым, скрыта от других модулей. Таким образом, при изменении реализации модуля нет необходимости вносить изменения в его клиенты, поскольку они используют лишь его интерфейс.

Приведенные выше принципы можно использовать не только в программировании, но и в процессе проектирования и разработки спецификаций. Эти вопросы подробно будут рассматриваться в главах 6 и 7. Сейчас же необходимо отметить, что преимущества объектно-ориентированного проектирования и анализа являются основными темами данной книги. Рассмотрение в следующей главе объектно-ориентированных языков программирования является основой для последующего изучения других глав.

Другие преимущества

Как подчеркивал Гради Буч (Booch), еще одним ключевым преимуществом объектно-ориентированного подхода является то, что с его помощью можно намного лучше управлять сложными системами [99]. Проблемы, моделируемые компьютерными системами, сами системы и управление процессом разработки — это действительно сложные процессы. Выделение ключевых элементов предметной области, объектная декомпозиция и наследование, объединение и использование структур, основанных на объектах и понятиях реального мира, позволяют управлять сложностью систем различными способами. Во-первых, система и проблема строго соответствуют друг другу, поэтому глубокое понимание повседневной жизни должно быть отражено в проектном решении. Во-вторых, инкапсуляция делит проблему на самостоятельные, небольшие и достаточно простые части, что способствует их пониманию как целого. В-третьих, объектно-ориентированный подход предоставляет несколько способов моделирования структуры и ее смысла: структуры наследования и т.д. И наконец, возможность повторного использования и расширяемость объектно-ориентированных систем означает, что сложные системы могут собираться из простых систем и последовательно развиваться в еще более сложные системы, проходя несколько простых этапов.

Более "открытые" и распределенные системы еще больше подчеркивают преимущества и необходимость применения объектно-ориентированных методов. Способность к взаимодействию и расширению, модульность, простые интерфейсы и сокрытие информации в значительной степени способствуют разработке действительно открытых систем. Принято считать, что открытые системы не должны привязывать покупателей или разработчиков к конкретным поставщикам технических средств, системного программного обеспечения или сетей. Чтобы добиться такого идеального результата в реальности, создаваемые системы должны взаимодействовать, т.е. они должны связываться с ресурсами и совместно их использовать. Хотя существует множество других мнений, вполне естественно рассматривать каждую систему, находящуюся внутри некоторой открытой распределенной системы, как объект, состоящий из других объектов. Каждая такая система должна быть связана не только с другими системами целиком, но и с их компонентами. Объектно-ориентированный подход поддерживает эту идею, предъявляя следующие требования к определению интерфейсов: они должны быть небольшими, заданными явным образом и в небольшом количестве. Задание протоколов взаимодействия, зависящих только от определения интерфейса (а не от реализации), — это, вероятно, единственный известный способ обеспечения работоспособности открытых систем. Поэтому вполне закономерна приверженность многих компаний открытым системам, которые вкладывают значительные средства в объектно-ориентированные технологии. Эти вопросы обсуждаются также сторонниками использования брокеров объектных запросов, позволяющих объектам или целым системам совместно функционировать в различных сетях. Предложенная группой OMG технология CORBA является стандартом для создания таких программных систем. Эта технология будет рассматриваться в главе 4.

Основной предпосылкой повторного использования является наличие библиотеки компонентов или хранилища программного обеспечения. Многие CASE-средства предоставляют такую возможность, однако они не могут обеспечить требуемую открытость хранилища или не удовлетворяют требованиям, предъявляемым к его компонентам. Эти требования обусловлены принципами использования лингвистических модульных единиц, простых интерфейсов и сокрытия информации. Преимущества объектно-ориентированных методов и объектно-ориентированного программирования могут принести еще большую пользу, если открытый код и/или библиотеки спецификаций будут широко доступны для коммерческого использования. Разработка таких библиотек, являющихся основой хранилищ программного обеспечения, является первоочередной задачей в области разработки программного обеспечения. Эти библиотеки станут еще более полезными, если будут связаны с реальными системами, разрабатываемыми в рамках коммерческих проектов. В качестве примера можно привести некоторые коммерческие хранилища, созданные компаниями Adaptive, Oracle и Unisys, однако они предназначены, в основном, для хранения метаданных и стандартов обмена файлами.

Несколько слов о компании AARDVARK

Задачи Aardvark при построении системы SACIS можно сформулировать следующим образом.

- Создать систему, которая одновременно была бы и базой данных, и системой обработки запросов
- Обеспечить возможность изменения и расширения системы в соответствии с изменениями требований бизнеса

- Обеспечить поддержку как обычных компонентов, так и компонентов экспертных систем
- Обеспечить возможность распределенной установки
- Не связывать компанию Aardvark с какими-либо продуктами других производителей
- Сохранять все компоненты, основанные на правилах, и семантические компоненты в явном виде, не скрывая их в коде, или в нормализованном виде
- Обеспечить возможность использования модулей, разработанных для системы SACIS, при разработке других систем

Хотя с самого начала было известно, что объектно-ориентированный подход является единственным решением, потенциально способным удовлетворить всем этим требованиям, менеджеры по информационным технологиям не были убеждены в том, что существуют некоторые средства разработки, которые позволят выполнить эту работу в реальности. Что необходимо предпринять специалистам компании? Консультанты Aardvark предусмотрительно рекомендовали следующий подход. Анализ технических требований и бизнес-правил следует проводить на основе объектно-ориентированных технологий. Это означает, что реализация на объектно-ориентированном языке программирования будет достаточно простой, если подходящий язык удастся найти к концу стадии анализа. В противном случае систему все же можно будет разработать на традиционном языке. При этом остаются все преимущества наличия спецификации расширяемых и повторно используемых компонентов. С учетом неопределенности выбора конечной платформы, для создания спецификации необходимо воспользоваться однократным прототипированием, для которого вполне естественно применить объектно-ориентированный язык. В качестве кандидата рассматривался язык Smalltalk, однако из-за имеющихся у программистов навыков и ввиду того, что прототип мог, кроме того, развиться в полнофункциональную систему, в конечном счете было решено использовать некоторую "смесь" языков Java и C++ вместе с соответствующими библиотеками, в том числе набор классов экспертной системы на языке Java. Для передачи сообщений на удаленные узлы при помощи броузера использовался язык HTML, а для поддержки связи с существующими приложениями и традиционными системами — язык XML. Разработчики получат базовые знания объектно-ориентированных концепций и языка Java, что поможет преодолеть любые предубеждения, сохранившиеся у них со времени работы на языке C. Проект начнется с трехдневного семинара по уяснению требований. Наряду с дальнейшей разработкой спецификации специалисты Aardvark изучают возможности по созданию компонентов баз данных. Этот проект будет рассматриваться и в последующих главах. Опасения специалистов Aardvark вполне обоснованны, поскольку объектно-ориентированная технология не лишена недостатков. В следующем разделе будут проанализированы некоторые из них.

2.2. Некоторые проблемы и заблуждения

Нужно внимательно оценивать преимущества любой технологии. Объектно-ориентированные методы — не панацея, поэтому важно определить области, где имеющиеся преимущества не могут быть достигнуты, а также указать те сферы, в которых эти преимущества можно достигнуть лишь при определенных условиях. Например, выше уже упоминалось, что повторное использование программного обеспечения является главным преимуществом.

В действительности же повторно используемые структуры данных продемонстрировали, что этой цели трудно достичь даже в том случае, когда используются объектно-ориентированные методы. Отчасти это можно объяснить тем, что проектирование таких модулей на практике приводит к возрастанию стоимости проекта, и здесь всегда существует соблазн получить готовый результат как можно быстрее, даже в ущерб возможности повторного использования. Ранее уже подчеркивалось, что подобный подход характерен для постиндустриального общества, особенно в периоды быстрого роста процентных ставок. В такой обстановке очень трудно выявить ситуацию, требующую дополнительных вложений в повторное использование. И хотя объектно-ориентированные методы сделали повторно используемые системы *менее* дорогостоящими, они не позволяют устранить всех дополнительных затрат. Дополнительные вложения могут быть оправданы только с точки зрения будущей экономии от повторного использования с учетом текущих цен. Это может оказаться серьезным препятствием на пути распространения такого обещанного и расхваленного преимущества. Однажды автору уже приходилось делать информационный обзор и обращаться к коллегам, участвующим в выполнении объектно-ориентированных проектов, с вопросом: какие повторно используемые компоненты были разработаны? При выборочном опросе участников около шести проектов только один специалист не заметил обсуждаемый эффект: "Ох! У нас действительно не было времени для выяснения этого вопроса. Нет, не было ничего, что мы могли бы когда-либо использовать снова". Однако в противоположность этому в следующем разделе будет рассказано о поразительных успешных стратегиях повторного использования. Более того, из главы 7 вы узнаете, как компонентно-ориентированная технология разработки способна быстро изменить всю картину. Вы увидите, что повторное использование должно быть спланированным и управляемым, точно так же как и проектирование компонентов, создаваемых для повторного использования в системах с другой архитектурой.

Еще одна проблема, которая уже упоминалась выше, заключается в отсутствии коммерческих объектных библиотек. Без таких библиотек классов и компонентов, разработанных в рамках реальных проектов, нельзя в полной мере воспользоваться преимуществами повторного использования. С другой стороны, почему же в условиях отсутствия таких библиотек какие-либо компании должны принимать эту технологию? Создается впечатление, что мы сами себя поставили перед дилеммой. Если новаторские компании не возьмут на себя значительную долю риска, руководствуясь какими-нибудь другими преимуществами, а не перспективами повторного использования, то полезные библиотеки компонентов не будут созданы и технология не будет развиваться. Успешность создания повторно используемых коммерческих компонентов зависит от готовности спонсоров проектов и инвесторов пойти на увеличение стоимости разработки и смириться с тем, что инвестиции будут возвращаться в течение длительного периода. В главах 6—9 будут рассматриваться аспекты повторного использования более высокого уровня: повторное использование результатов проектирования и спецификаций. Здесь также пригодятся хорошие библиотеки и средства работы с ними.

Для того чтобы библиотеки были полезными, нужно научиться справляться с их объемом и сложностью. Однако даже при существовании таких объектных библиотек, а также разработанных для них и широко распространенных соответствующих средств классификации и навигации возникает другая проблема. Каким образом разработчики могут узнать, что конкретная библиотека содержит именно тот объект, который им нужен в конкретном случае? Здесь возникают трудности, связанные с огромным количеством информации. Люди просто не в состоянии предположить, что находится в каждой библиотеке. Невозможно даже представить, чтобы каждая библиотека содержала все (или почти все) объекты, которые могут понадобиться обычному разработчику. Возможны разные пути решения этой проблемы.

Специализированные обслуживающие компании могут выполнять роль информационных брокеров. Экспертные системы, содержащие подобную информацию, могут разрабатываться и распространяться на коммерческой основе. Потребуется также разработать механизмы группирования объектов и принципы их классификации. Для описания взаимодействия объектов в библиотеке понадобятся языки высокого уровня. Широкое применение библиотек может так никогда не продвинуться дальше использования библиотек, содержащих только программные и интерфейсные объекты, которые аналогичны существующим в настоящее время. Вполне возможно, что преуспевающий поставщик займется созданием одного большого продукта, в котором будет содержаться большая часть полезных классов других разработчиков, однако подобная перспектива выглядит весьма отдаленной. Повторное использование элементов в различных предметных областях может быть успешным лишь в случае разработки классов, не зависящих от области их применения.

Автор допускает, что привязанные к определенной предметной области библиотеки будут одним из аспектов деятельности компаний, занимающихся консалтингом и разработкой. Такие библиотеки будут использоваться в заказных проектах. Однако эти компании (по крайней мере, консалтинговые), как правило, не расположены делать инвестиции такого рода, хотя в сфере финансов уже имеются определенные предложения, и группа OMG предлагает создать новые стандарты услуг в отдельных секторах промышленности.

Управление большими библиотеками компонентов для объектно-ориентированного программирования в значительной степени представляет собой все еще неразрешимую проблему. Были разработаны два основных способа организации таких библиотек. Классы могут быть организованы иерархически, что для просмотра позволяет использовать броузер. В соответствии с другим вариантом поиск и извлечение данных могут производиться на основе ключевых слов, хранящихся вместе с классами, как в обычных системах обработки данных. Оказалось, что ни один из подходов не является идеальным, и в настоящее время наилучшим считается комбинированный способ. Однако еще не существует достаточно развитых программных продуктов, облегчающих решение этой проблемы. Поэтому в большинстве случаев используются "ручные" методы.

Упомянутая выше проблема, связанная с обработкой непредвиденных ситуаций, является еще одним серьезным препятствием на пути к широкому распространению объектно-ориентированных методов. Повторное использование объекта может оказаться под угрозой, если какой-нибудь усердный программист обнаружит в нем достаточно грубую или небезопасную ошибку. *И рушится сквозь потолок на нас нужда*[1]. Для частичного решения этой проблемы можно воспользоваться средствами управления конфигурацией, однако, как и в случае с кодогенераторами CASE-средств, далеко не всегда можно воспользоваться средствами управления во время неправильного функционирования действующей системы.

Кроме того, необходимо разработать протоколы внесения изменений и контроля версий в системах, состоящих из сложных объектов со множественными взаимосвязями. С целью разработки языков компоновки потребуется разработать формальное описание интерфейсов объектов. Это позволит настроить конфигурацию библиотечных модулей в соответствии с требованиями спецификации. Достаточно интересным является вопрос о создании средств, призванных оценить влияние измененного интерфейса на другие объекты системы [394].

Большая часть объектно-ориентированных языков программирования не поддерживает концепцию постоянных объектов, т.е. объектов, которые хранятся на диске и остаются неизменными до следующего выполнения программы. Таким образом, управление данными и

---

[1] *Строка из стихотворения Бернса.* — Примеч. пер.

объектами поддерживается объектно-ориентированным программированием не очень хорошо. Именно этим недостатком вызвано появление объектно-ориентированных баз данных, однако все же еще остается большой разрыв между специалистами, занимающимися объектно-ориентированным программированием, и специалистами, которые разрабатывают объектно-ориентированные базы данных. И в самом деле некоторые пуристы, такие как Вегнер (Wegner), заявляют, что постоянные объекты противоречат основной философии объектно-ориентированного подхода. По мнению автора, стиль программирования, не допускающий постоянство существования объектов, просто обречен. Объектно-ориентированные базы данных будут подробно рассматриваться в главе 5.

Эффективность некоторых объектно-ориентированных языков программирования, поддерживающих динамическое связывание и механизм сборки мусора (см. главу 3), вызывает сомнения. Для реализации некоторых приложений (возможно, внедренных или работающих в реальном времени), для которых эффективность функционирования или их размер имеют первостепенное значение, эти языки просто нежизнеспособны, хотя на них были разработаны некоторые небольшие приложения реального времени. С другой стороны, иногда могут оказаться более эффективными гибридные языки, однако зачастую они нарушают принципы и не позволяют воспользоваться преимуществами объектно-ориентированного подхода. Такой же парадокс наблюдается в области экспертных систем, которые одно время часто разрабатывались на таком неэффективном языке, как Lisp. В настоящее время проблема успешно решена, и появились достаточно продуманные гибридные средства разработки. Такое развитие средств объектно-ориентированного программирования должно предшествовать широкому использованию преимуществ этого подхода.

В главе 1 было показано, что наследование затрудняет повторное использование, если строго не придерживаться определенных процедур. Таким образом, по мнению некоторых сторонников компонентно-ориентированной разработки и других специалистов, методы, в которых не используется наследование, занимают особое место в арсенале разработчика программного обеспечения.

Выше уже неоднократно подчеркивалась выгода от уменьшения затрат на поддержку, получаемая при правильном объектно-ориентированном проектировании и программировании. Но даже в этом случае существует опасность выполнения необдуманных действий. На стадии проверки объект должен подвергаться тестированию и демонстрировать надлежащее поведение вне своих границ. Кроме того, должны рассматриваться целиком и все соответствующие структуры наследования. Это означает, что при использовании объектно-ориентированных методов программные продукты должны тестироваться более тщательно и на более ранних стадиях выполнения проекта.

Повторное использование некоторых частей приложения по-прежнему требует больших творческих способностей. Разработчикам нужно выделить функции из различных предметных областей и отделить их от функций реализации приложения. Точно так же нельзя сформулировать исчерпывающий набор правил для автоматической обработки как таковой и переквалификации персонала. Плохой программист может без труда создавать бесполезные, абсолютно непригодные для повторного использования системы как при помощи объектно-ориентированного языка, так и при помощи любого другого. Как гласит старая поговорка, настоящие программисты пишут на FORTRAN, но могут делать это и на *любом* другом языке.

Структуры наследования представляют собой лишь одну форму семантики. Как станет видно из последующих глав, существуют и другие семантические формы, включающие взаимодействие на платформе "клиент/сервер", ассоциативные и компоновочные структуры,

ограничения, накладываемые на количество элементов, бизнес-правила и правила управления. По мнению автора, включение в систему любой семантической информации может уменьшить возможность повторного использования ее компонентов. Причина в том, что чем больше сообщается о чем-либо, чем больше что-нибудь конкретизируется, тем больше это нечто становится зависимым от контекста. С другой стороны, не слишком конкретный объект будет плохой моделью объекта реального мира, который всегда воспринимается и используется в определенном контексте. Разработчики компонентов, аналитики и программисты должны искать компромисс между повторным использованием и семантикой. В некоторых приложениях, например в графическом пользовательском интерфейсе, где сами объекты, по существу, являются искусственными артефактами и контекст сильно ограничен, эта проблема стоит не так остро. В коммерческих системах общего назначения и даже в области проблемного моделирования это далеко не так.

Как уже отмечалось, для отладки системы иногда может понадобиться проанализировать ее поведение, отслеживая передачу сообщений от одного объекта к другому. Такой анализ с целью отладки или осмысления функционирования системы может оказаться невероятно сложным и остается, по мнению автора, очень серьезным недостатком объектно-ориентированных систем. Можно представить себе, как будущие поколения разработчиков программного обеспечения будут критиковать своих теперешних коллег за "структуры передачи сообщений, реализованные в виде спагетти" или нечто подобное.

Разработки в этой области ведутся и в настоящее время, хотя и с меньшей интенсивностью, и некоторые современные языки (даже Java!) могут быть вытеснены более новыми. Например, полная поддержка множественного наследования, значение которого было определено в предыдущей главе, в любой отдельно взятой системе пока еще реализована. Другими словами, ни одна система, имеющая широкое коммерческое распространение, для разрешения конфликтов не использует наследование (на уровне классов или экземпляров) методов, атрибутов и значений атрибутов с явно заданными пользовательскими правилами. Не существует также объектно-ориентированных языков программирования, поддерживающих динамическую или множественную классификацию. Поддержка множественного наследования и универсальность отсутствуют даже в таких популярных языках, как Java. По моему мнению, это большой недостаток. Остается надеяться на появление новых языков с более широкими возможностями, призванных решить эту проблему. С точки зрения построения больших систем на основе объектно-ориентированного языка программирования такая возможность выглядит достаточно тревожно, поскольку организации с трудом представляют себе возможность перехода на другие языки в процессе выполнения какого-либо проекта. К счастью, эта проблема не затрагивает объектно-ориентированный анализ или проектирование.

Выше уже подчеркивались преимущества использования простых интерфейсов и сокрытия информации. При этом утверждалось, что изменение реализации модуля не будет отражаться на его клиентах. Однако иногда это совсем не так. Предположим, что изменения настолько существенны, что изменяется и сам интерфейс. Допустим, мы достаточно комфортно себя чувствовали, используя понятие "служащий", и все наши системы и базы данных содержали единообразный интерфейс для обращения к объектам служащих. Теперь вводится самоуправление трудящихся, рабовладение или некоторая другая совершенно иная социальная система. Конечно, теперь наше понятие интерфейса объекта служащего должно подвергнуться радикальному изменению. Каким образом мы должны теперь создавать служащих, которые, например, не получают зарплату, или акционеров, не получающих прибыли?

Библиотеки функций, особенно при программировании в научных исследованиях, достаточно хорошо соответствовали стратегии повторного использования, однако зачастую их применение было затруднено из-за необходимости запоминания длинных списков параметров или "общих блоков" в FORTRAN. В некоторых случаях требовалось иметь представление о внутренней работе таких функций. Осталось только узнать, будут ли будущие поколения программистов точно так же критиковать объектные библиотеки? Исходя из высказанных в предыдущем разделе соображений, я полагаю, что нет.

Основные дополнительные затраты, связанные с переходом на объектную технологию, — это инвестиции в новые технические/программные средства и методы, которые могут понадобиться. К этим затратам можно отнести также стоимость необходимого обучения и переквалификации как разработчиков, так и управляющего персонала, а также дополнительные затраты на разработку повторно используемых компонентов и управляющих библиотек. Данная тема рассматривается в главе 9.

В таких условиях стоимость написания повторно используемых компонентов может оказаться намного выше. В работе [417] предполагается десятикратное увеличение, а жизненный опыт подсказывает, как минимум, шестикратное. Для некоторых приложений, например графических пользовательских интерфейсов или приложений из финансовой сферы, библиотеки классов или контуры приложений могут распространяться на коммерческой основе. Это, по меньшей мере, исключает затраты на разработку компонентов. Однако в настоящее время такой подход применим лишь в немногих предметных областях. После написания или приобретения набора классов эти проблемы не исчерпываются. Библиотеки по управлению компонентами являются очень сложными и дорогостоящими. Как вы увидите ниже, определение библиотек бизнес-объектов и управление ими, а также контроль версий компонентов все еще остаются проблемами, не имеющими известных общих решений. Хотя стоит заметить, что эта прикладная область все же развивается.

Одна из самых сложных проблем состоит в организационных и культурных изменениях, которые непременно последуют за переходом на объектную технологию, если она станет выгодной. Обычно аналитики и программисты вознаграждаются за количество созданного ими кода (конечно, при этом учитывается то поколение языка программирования, на котором они работают), а не за то количество кода других разработчиков, которое они могут повторно использовать. Такое положение дел подразумевает изменение самой системы поощрения, которая может оказаться большим препятствием. Более того, руководителям проектов платят деньги за своевременное выполнение самих проектов, а не за написание кода с целью извлечения выгоды из последующих проектов, что также является большим препятствием. Некоторые специалисты считают, что разработка классов должна быть полностью отделена от создания приложений. Однако это зачастую непрактично и не всегда выполнимо, поскольку очень много идей формулируется при непосредственном участии пользователей.

И последняя, относящаяся к объектной технологии проблема, которая будет рассмотрена, связана с тем, что наиболее ярые сторонники этой технологии считают ее самым последним словом в области разработки программного обеспечения, так называемой "серебряной пулей". Крупным инвестициям в объектно-ориентированную технологию может помешать более здравое суждение, согласно которому эта технология в конечном счете окажется еще более сильной метафорой, чем реляционные базы данных, когда-то также провозглашенные последним достижением, а теперь заменяемые в некоторых прикладных областях более новыми технологиями (см. главу 5). Признаком существования других технологий (не объектно-ориентированных) может служить появление новых моделей, основанных на исследовательских языках типа BETA (см. раздел 3.6.1). К счастью, оказалось,

что подобные разработки будут дополнять, а не вытеснять объектно-ориентированную парадигму. Это позволит защитить вложенные к настоящему моменту инвестиции от изменения технологий.

2.3. Примеры

В этом разделе будут очень кратко проанализированы некоторые преимущества объектно-ориентированной технологии, о которых сообщалось в различных источниках.

Возможно, самым первым и наиболее хорошо известным случаем крупномасштабного успешного применения объектно-ориентированной технологии можно считать ее использование компанией Brooklyn Union Gas. Эта компания заменила систему управления клиентами, содержащую 1,5 миллиона строк кода на языке PL/1, другой системой, написанной с использованием объектно-ориентированного препроцессора. Новая система была очень большой. Она была рассчитана на 850 интерактивных пользователей, включала базу данных размером 100 Гбайт и 10000 модулей кода. Согласно отзывам, выгода от использования этой системы заключалась в уменьшении размеров кода на 40% благодаря, главным образом, повторному использованию, снижению затрат на поддержку (группа из 12 человек), безотказному вводу в действие и, прежде всего, очень большой гибкости и возможности расширения. Эти преимущества не были бесплатными. Разработчикам пришлось изобретать собственные объектно-ориентированные методы разработки и стандарты, поскольку в период с 1987 по 1990 гг. соответствующие спецификации еще не были созданы, а традиционных методов оказалось недостаточно.

Еще одним примером успешного применения объектно-ориентированной технологии на ранней стадии ее развития может послужить система управления техническим обслуживанием компании General Motors. Эта система помогала составлять график ремонтных работ и позволяла выполнять инвентаризацию запасных частей. В старой системе содержалось 265 тысяч строк кода на языке PL/1. Эта система разрабатывалась в течение 12,5 человеко-лет и при работе использовала 13,6 Мбайт оперативной памяти. Новая система была разработана на языке Smalltalk 80 менее, чем за один человеко-год и содержала всего лишь 22 тысячи строк кода, а также использовала не более 1,1 Мбайт оперативной памяти. Примечательно то, что рабочие характеристики обеих систем были примерно одинаковы, а общий выигрыш в производительности можно было оценить соотношением 14:1. Конечно, повторно создавать систему почти всегда намного проще, однако это соотношение все равно впечатляет.

Хотя выше утверждалось, что добиться повторного использования (по сравнению с возможностью расширения) очень трудно, некоторые компании все же смогли реализовать это. Компания Petroleum Information предоставляет географические данные для предприятий нефтяной и газовой промышленности. В этой компании с использованием системы CLOS (системы объектно-ориентированного программирования на языке Common Lisp) была создана геоинформационная система Sorcerer's Apprentice (т.е. "ученик чародея"), о которой сообщалось, что 80% ее исходного кода используется повторно. Это, в свою очередь, привело к существенному сокращению сроков разработки и попыткам со стороны разработчиков преодолеть те проблемы, которые ранее считались трудноразрешимыми.

К другим крупномасштабным приложениям относятся многие торговые системы, разработанные для банковской сферы. В начале 1990-х годов в корпорации Swiss Bank использовался язык Objective C. При этом на протяжении трех лет удалось добиться 50%-го увеличения производительности программистов [326].

В [362] описывается 20 из 225 исследованных конкретных случаев наиболее успешного применения объектной технологии, которые были выдвинуты на конкурс *Object World* (Мир объектов) в 1992 году. К ним относятся бухгалтерская и административная система компании Southern California Gas, моделирующая программа с применением объектно-ориентированного анализа компании Boeing и набор средств разработки программ для лечебных учреждений. Присуждение наград продолжается, и о них можно узнать из трудов многих конференций *Object World*.

Позднее традиционные системы были быстро реконструированы с использованием объектно-ориентированного подхода и связанных с ним методов, таких как экстремальное программирование (eXtreme Programming) [63]. Автор и многие его коллеги участвовали в различных успешных объектно-ориентированных проектах, причем успех достигал таких масштабов, что становился банальным и привычным. Однако это совсем не означает, что на этом пути не было многочисленных провалов. Примечательно то, что многие из возникающих трудностей заставляли организации, которые в начале 1990-х годов приняли язык программирования C++ и связанные с ним методы, отказаться от объектной технологии.

Как показывает накопленный опыт, повторное использование вполне достижимо, однако для этого необходимо гораздо больше, чем просто хороший язык объектно-ориентированного программирования. Для этого требуются соответствующие средства управления проектом, надежные методы и, кроме всего прочего, пристальное внимание к проблемам образования, обучения и управления изменениями, а также немного везения. Тогда можно добиться повторного использования, но только при определенных затратах. Конечно, для написания и тестирования повторно используемого кода требуется несколько больше времени. Однако непосредственные преимущества расширения зачастую оттесняют выгоды от повторного использования на почетное второе место. Расширяемость является одной из главных характеристик систем, востребованных в сфере бизнеса. Таким образом, даже если повторное использование не приемлемо, от применения объектно-ориентированных методов можно все же получить большой выигрыш.

Несмотря на многочисленные успехи, в настоящее время ситуация все же складывается так, что подавляющее большинство организаций не в состоянии широко воспользоваться преимуществами повторного использования. В основе этого лежат, скорее всего, экономические и административные причины, которые непосредственно не связаны с самой технологией (они рассматриваются в главе 9).

2.4. Стратегии перехода

Риск, связанный с новыми технологиями, состоит в том, что люди либо слишком увлечены этими технологиями и нереалистичны в своих ожиданиях, либо слишком пессимистичны и не склонны переходить на новые технологии. В первом случае люди страстно и беспечно "окунаются" в технологию и зачастую ею совершенно не руководствуются, если вообще не разрушают свою карьеру. Во втором случае луддитов опережают компании, которые сразу оценивают наилучшие аспекты новой технологии, используют ее и получают преимущество перед конкурентами.

В данном случае примером могут служить экспертные системы. В начале 1980-х годов, когда экспертные системы вышли из стен лабораторий, занимающихся исследованиями в области искусственного интеллекта, их сопровождала громкая реклама в средствах массовой информации, которая во многом предоставляла искаженную информацию. Однако многое

было взято из пропаганды, проводимой специалистами по искусственному интеллекту, которым следовало бы расширить свои знания, однако они лишь стремились добиться финансирования на выполнение своих исследовательских работ. Таким образом, нам досталась достаточно незрелая технология с ограниченными реальными возможностями. Поэтому на рынке по-прежнему "ожидается" появление систем, которые были бы "способны выполнять любую задачу, которую человек может выполнить в течение 20 лет" [713]. Это и есть формула провала — неразвитая технология плюс нереалистичные ожидания.

Провал подобного процесса можно карикатурно представить в виде следующей притчи. Менеджер отдела по обработке данных некоторой большой корпорации, скажем, в 1983 году прочитал в прессе об экспертных системах и воодушевился их возможностями. Эксперту по научно-исследовательским и опытно-конструкторским работам было поручено исследовать этот вопрос. Предварительные исследования показали, что можно получить определенные преимущества, и было принято решение запустить проект. До сих пор все было хорошо, но с этого момента события начинают развиваться по неправильному пути. Менеджер отдела обработки данных (назовем его Иван Петров), анализируя, кому можно было бы поручить выполнение этого задания, оценивает специалистов корпорации и находит отдел исследования операций. До настоящего времени этот отдел не делал ничего полезного уже в течение многих лет (это подразделение продемонстрировало свою несостоятельность в конце 1960-х годов благодаря процессу, подобному описываемому сейчас). Итак, Иван пришел к довольному (от перспектив быть полезным) руководителю отдела исследования операций, который сразу же предложил свою помощь. Кому бы поручить выполнение задания? Хорошо, ребята из отдела С именно сейчас ничем не заняты, и таким образом был создан сектор экспертных систем. Для нового подразделения было выделено минимальное финансирование, и для ознакомления была куплена пара оболочек экспертных систем. Его специалистам было поручено найти подходящую проблему для экспертных систем и сообщить об успешности ее решения. Время шло, и сектор экспертных систем постепенно отстранили от всех приложений, имевших хоть какое-нибудь значение для бизнеса: "Что если это будет работать неправильно? Люди заметят". Если они находили для себя приложение, возможно, связанное со "справочным столом",— решение не работало достаточно успешно, но и это не имело значения, поскольку никто ничего существенного не отмечал. Однако после пяти или шести попыток Иван отметил, что изысканий вполне достаточно, и пришел к выводу, что экспертные системы были излишне разрекламированы. На самом деле он использовал для этого немного другие выражения, но мы не будем здесь на них останавливаться. В этой компании (как и во всем мире в целом) несбывшиеся надежды дали повод для пессимизма, и в конце 1980-х годов началась так называемая "зима" искусственного интеллекта. Экспертные системы, функционирующие в настоящее время, это действительно развитые, мощные и полезные средства, и многие компании получают существенные выгоды от их использования. Однако эти компании имеют совершенно других менеджеров по обработке данных, не таких как господин Петров.

Хочется надеяться, что к этому моменту уже вполне понятно, что экспертные системы сравниваются с объектно-ориентированным подходом. Последний представляет собой попрежнему достаточно новую технологию, и существует широко распространенное мнение, что этот подход может быть решением многих проблем в сфере информационных технологий. Как следует действовать компаниям, желающим поэкспериментировать с объектно-ориентированными системами? Ответ прост: не делать то, что делал Иван. Вот некоторые советы, которыми стоит воспользоваться.

- Для команды разработчиков выберите людей, которых вы только можете себе позволить. Объектно-ориентированные методы, точно как и экспертные системы, требуют очень хороших специалистов, а не останков с кладбища слонов корпорации.
- Выберите очень важное для коммерческой деятельности приложение, чтобы протестировать на нем методы, но сделайте поправку во времени на возможные неудачи. Обязательно выберите небольшое приложение и убедитесь в том, что оно выдало предупреждение, ошибку или завершилось успешно.
- Выделите бюджет в соответствии со значимостью приложения. Это подразумевает правильное расходование денежных средств и обоснование принимаемых решений.
- Определите четкие цели для группы разработки, не только технические, но и цели, связанные с коммерческой деятельностью.
- При выполнении проекта четко определите механизм передачи разработанных фрагментов основным пользователям. Постоянно держите пользователей в курсе всех событий и отчитывайтесь о том, какие результаты получены.

Из сказанного можно извлечь некоторые общие уроки. Они применимы почти ко всем новым технологиям. Когда новая технология разрабатывается впервые, ее потенциальные возможности невелики и она почти неизвестна на рынке. Постепенно, по мере вступления нового подхода в жизнь, эти возможности будут возрастать, пока не будет достигнут некоторый уровень насыщения или ее потенциальные возможности не истощатся. Однако приобретаемая технологией популярность приводит к формированию субъективного восприятия, вызванного громкой рекламой независимо от реальных возможностей технологии. Это иллюстрируют кривые, изображенные на рис. 2.6.

И наконец, как только активная реклама затихает и первые пользователи опробуют технологию, наступает депрессия. Обнаруживается, что существует множество проблем и чрезмерно преувеличенные заявления как поставщиков, так и журналистов не имеют ничего общего с реальностью. В этой точке кривая ожиданий начинает падать до уровня первоначальных реальных возможностей. Однако тем временем технологии совершенствуются, и две кривые на диаграмме пересекаются в точке Q. Теперь цена базируется на рыночной стоимости, отражающей спрос, который, в свою очередь, базируется на ожиданиях, а не на потребительской стоимости или характеристиках. Таким образом, покупатели, приобретающие технологию в период между точками P и Q, должны переплачивать за нее. После точки пересечения они могут приобретать технологию со скидкой до тех пор, пока нормальная стоимость не восстановится в точке R.

Единственная сложность использования этой модели связана с определением конкретной точки развития технологии на кривой ожиданий. Техническая литература является хорошим индикатором ожиданий, и эксперты в области новых технологий обычно в состоянии оценить их реальные возможности. Возможно, в момент написания этой книги (середина 2000 года) мы находимся непосредственно на отрезке Q-R. Это касается объектно-ориентированного программирования, объектно-ориентированных методов и объектно-ориентированных баз данных примерно в равной степени, хотя объектно-ориентированное программирование более развито, чем две другие сферы. Поскольку для полного осмысления и принятия нового подхода любой организации требуется примерно два года, именно сейчас для пользователей наступило время перехода на объектную технологию: пока размеры скидок все еще приемлемы.

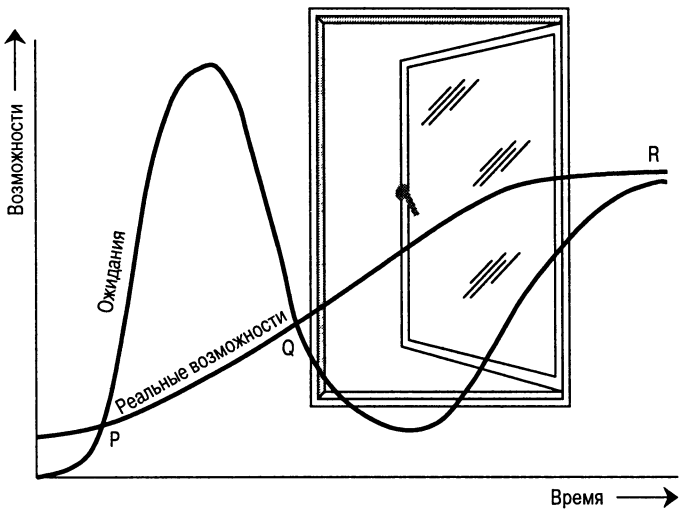


Рис. 2.6. Возможность перехода на новые технологии

Опыт автора свидетельствует о том, что тенденции рынка и их характер изменяются. В 1990 году представители организаций посещали информационные совещания и собирали информацию. К 1991 году возможности объектной технологии, предлагаемые поставщиками, стали играть важную роль и претендовать на более широкое использование в области вычислений. Но лишь в немногих проектах, призванных решать ответственные задачи, начала использоваться объектная технология. Рынок услуг по обучению объектно-ориентированному программированию и объектно-ориентированным методам начал развиваться в 1993 году, и самые смелые пользователи завершали свои первые пробные программы. В 1993 году объектная технология стала использоваться в первых крупномасштабных и очень важных проектах по обработке данных. И лишь немногие продвинутые пользователи полностью переключились на объектно-ориентированную разработку новых систем, поддерживая одновременно с этим уже существующие действующие системы, ожидающие пересмотра с учетом новых технологий. Один из руководителей отдела информационных технологий корпорации Swiss Bank признал (*Computing*, 19 ноября 1993 г.), что основным мотивом для перехода послужила возможность быстрой разработки системы. Объектная технология способствует быстрой разработке различными способами, но, главным образом, благодаря возможности повторного использования и расширения. Комплексный подход к быстрой объектно-ориентированной разработке приложений представлен в главах 6—9. К 1997 году объектная технология стала преобладающей практически во всех организациях, связанных с информационными технологиями. Ею пользуются почти все компании-разработчики, хотя полный переход замедлился из-за проблемы 2000 года и ввода евро. Практически не вызывает сомнений тот факт, что первые годы XXI века ознаменуются почти глобальным переходом на объектную технологию.

Специалисты по управлению должны очень хорошо представлять себе все преимущества и последствия перехода на объектную технологию, а также нельзя забывать о подводных камнях, неожиданно обнаруживаемых при переходе к этой технологии. Многие организации предъявляют к объектной технологии особые требования, и зачастую наилучшим выходом

является специально разработанная программа обучения. Преподавание как средство передачи знаний всегда имело большой успех. Это означает, что с группой разработчиков будут работать опытные специалисты, которые окажут помощь и передадут свой практический опыт.

Основная опасность заключается в том, чтобы не рассматривать объектную технологию как "серебряную пулю". Следует соблюдать осторожность, однако хочется верить, что современные компании будут в состоянии избежать этой опасности в ближайшие годы, поскольку новый подход абсолютно отличается от той деятельности, которую я люблю называть "построением компьютерных систем".

Возможно, главная проблема, с которой сталкиваются менеджеры, начинается с вопроса, *где они теперь оказались со своим наследием, состоящим из обычных, связанных с трудностями поддержки, но жизненно необходимых систем*, и заканчивается другим вопросом, *где они хотят оказаться, взяв на вооружение объектную технологию*. Эта проблема связана со стратегией перехода, которая подробно рассматривается в разделе 4.4.

Перед тем как организация сможет всерьез рассмотреть возможность перехода на объектную технологию, сначала нужно определить, где же она находится в настоящее время: какие классы средств, программных продуктов, методов и языков программирования используются. Зачастую полезно убедить программистов, работающих на языке COBOL, в том, что понятия объектной технологии им уже знакомы и многие принципы объектно-ориентированного проектирования также хорошо им известны как "практика хорошего проектирования программного обеспечения". К ним относятся принципы модульности, сокрытия информации, слабого связывания и т.д. Это поможет принять любые необходимые изменения без особого труда. Следует также обратить их внимание на то, что традиционный модуль можно рассматривать как набор интерфейсных функций, совместно использующих структуру данных.

Организации при переходе на объектную технологию должны понимать, что данная технология имеет как преимущества, так и недостатки. К преимуществам относятся удобство и простота использования, быстрая разработка, более простое изменение проектных решений, возможность работать с более сложными системами, повторное использование на более высоком уровне и повышенная гибкость. Семантическая выразительность объектных моделей делает описание более обратимым и влияет на скорость разработки приложений. Возможность расширения также предоставляет непосредственные преимущества. Полный объектно-ориентированный анализ позволяет извлечь пользу из объектной технологии на самых ранних стадиях жизненного цикла системы. Повторное использование позволяет сократить стоимость разработки. Благодаря этому некоторые организации ежегодно удваивали производительность на протяжении нескольких лет. Недостатки заключаются в том, что повторное использование требует дополнительных затрат, а его успешное применение может быть связано с большими социальными изменениями и очень крупными дополнительными инвестициями. Организации, переходящие на объектную технологию, должны будут кардинально изменить систему вознаграждения, причем специалисты в области повторного использования будут поощряться совершенно иначе, чем разработчики, отдельно вознаграждаемые за скорость и качество. Ведущие специалисты не должны поддерживать ранее созданные важные системы, поскольку их обширные познания в области бизнеса должны приносить пользу в рамках новых объектно-ориентированных разработок. Каждый менеджер организации должен будет пройти полную переподготовку в соответствии с новым подходом, в противном случае можно снова вернуться к кошмарам каскадной модели разработки и структурированному представлению при помощи объектно-ориентированных средств. Это требует четкого, уверенного

руководства и абсолютной ясности, а также четкого осознания преимуществ и рисков, связанных с применением объектно-ориентированной технологии.

Диапазон приложений, создаваемых с помощью объектной технологии, практически не ограничен. Анализ объектно-ориентированных систем позволяет создавать семантически выразительные модели и способствует выполнению проектов, направленных на изменение условий ведения бизнеса.

В сфере компьютерных технологий объектная технология требуется для решения проблем пользователей, участвующих во все более динамичном бизнес-процессе. Ключом успешного перехода является комплексный подход к объектной технологии на протяжении всего жизненного цикла систем: от начального этапа определения требований до их реализации. Истинной причиной принятия объектно-ориентированного подхода является возможность разработки гибких, дружественных и надежных систем. Это позволяет решать проблемы поддержки посредством создания более точных спецификаций, основанных на моделировании с использованием семантически выразительных языков. Решение этой задачи облегчается также благодаря обеспечению более простой расширяемости, основанной на наследовании и компонентном проектировании. Это способствует как разработке новых систем, поскольку их можно создавать из повторно используемых компонентов, которые инкапсулируют состояние и поведение абстракций, связанных с реальными объектами из предметной области, так и созданию более безопасных систем с помощью более полного использования свойств инкапсуляции состояния объектов. Кроме того, это позволяет создавать дружественные и гибкие пользовательские интерфейсы, поддерживает модульность (повторное использование) на очень высоком уровне и обеспечивает простоту расширения набора функций. Хотя все представленные выше преимущества иногда могут быть использованы не полностью, например из-за некоторого противоречия между абстракцией и наследованием (см. главу 1), причины все же остаются убедительными для тех компаний, которые в условиях жесткой конкуренции желают добиться значительных успехов.

Все сказанное выше приводит к необходимости формулировки весьма жесткого требования: принимать объектно-ориентированные методы нужно очень осторожно, но вместе с тем и достаточно активно. Развитие объектно-ориентированных методов достигло той стадии, когда можно воспользоваться их широкими возможностями.

2.5. Резюме

Эта глава посвящена, в основном, реальным преимуществам, которые могут быть получены в результате применения объектно-ориентированного программирования и, в меньшей степени, объектно-ориентированных методов в целом. Преимущества объектно-ориентированного анализа и проектирования более подробно рассматриваются в главах 6 и 7. В главе 5 будут подробно обсуждаться преимущества объектно-ориентированных баз данных. Кроме того, вы познакомитесь и с некоторыми недостатками объектно-ориентированного подхода.

Преимущества можно подытожить следующей формулой.

```
инкапсуляция + наследование + идентичность = повторное использование +
расширяемость + семантическая выразительность
```

Повторное использование и расширяемость, а также другие преимущества объектно-ориентированного подхода применимы не только в программировании, но и в процессе проектирования и разработки спецификаций.

Декомпозиция "сверху вниз" может привести к созданию модулей, зависимых от приложения, и сделать невозможным повторное использование. Проектирование "снизу вверх" и принцип сокрытия информации максимально повышают возможность повторного использования. Этому способствует также и инкапсуляция.

Полиморфизм и наследование упрощают отслеживание вариаций и исключений и, следовательно, позволяют создавать более гибкие системы. Принцип открытости-замкнутости реализуется с использованием наследования. Наследование обеспечивает возможность расширения, но вместе с тем может затруднить повторное использование.

Семантическая выразительность также связана с наследованием и другими естественными структурами, которые вместе с ограничениями и правилами определяют контекстное существование объекта. При этом с повторным использованием также могут возникнуть проблемы, поэтому вопросам управления необходимо уделять самое пристальное внимание.

В данной главе были также рассмотрены пять критериев и пять принципов объектно-ориентированного проектирования, сформулированных Мейером (Meyer).

Кроме того, упоминались следующие преимущества.

- Поддержка выполняется локализовано. Поэтому она требует меньших затрат и меньше подвержена ошибкам.
- Обеспечивается более высокая производительность и одновременно с этим более высокое качество.
- Наследование позволяет создавать более гибкие и легко расширяемые системы, поддержка которых требует меньших затрат.
- Объектно-ориентированный подход является средством управления сложными системами. Объектная декомпозиция систем способствует решению проблемы их масштабируемости.
- Распределение работы между разработчиками происходит более естественно и выполняется гораздо проще.
- Улучшенная возможность моделирования и эволюции систем.
- Описание интерфейсов между внешними и ранее существующими системами значительно упрощается.
- Переход от концептуального моделирования к программированию становится еще более незаметным.
- Объектно-ориентированные модели способны лучше отразить суть приложения.
- Инкапсуляция способствует созданию безопасных систем.
- При необходимости можно использовать формальные методы создания спецификаций.
- Для некоторых приложений объектная технология не имеет равноценной альтернативы.

Приведенные преимущества не являются безоговорочными. Объектно-ориентированный подход имеет ряд "подводных камней".

- Сжатые сроки выполнения проектов могут означать, что возможность повторного использования не гарантируется.
- Должны быть разработаны библиотеки и распространены сведения о них.
- Данная область быстро развивается, и современные языки могут быть заменены другими. Например, полная поддержка множественного наследования и универсальность отсутствуют даже в таких популярных языках, как Java.
- Развитие рынка компонентов еще продолжается, и его еще рано считать устоявшимся.
- При использовании наследования или других семантических конструкций для обеспечения возможности повторного использования необходимо осуществлять строгое управление.
- Такие вопросы, как поддержка постоянных объектов, параллелизм и эффективность, все еще ожидают своего решения.
- Большое значение имеет топология передачи сообщений. Так что вполне допустимо написание плохих объектно-ориентированных систем.
- Написание повторно используемых компонентов может оказаться гораздо дороже, чем ожидалось.
- Управление библиотеками компонентов вызывает большие трудности и связано со значительными затратами.
- Необходимы культурные изменения, которые никого не могут привести в восторг.
- Неизбежны затраты на обучение и переподготовку специалистов.
- Объектная технология может оказаться и не последним словом в области разработки программного обеспечения.

Некоторые специалисты, принимавшие участие в коммерческих проектах, сообщали о реально достигнутых преимуществах, в том числе об уменьшении размеров исходного кода, простоте поддержки, больших возможностях повторного использования и хорошей расширяемости. Существует широко распространенное мнение, что, по сравнению с обычными методами, объектно-ориентированный подход способствует получению гораздо большего эффекта, однако на данный момент это подтверждается лишь немногими эмпирическими доказательствами. Небольшое число достаточно крупных проектов действительно подтверждает это. Правда, это касается лишь масштабируемости. В некоторых организациях имеются управленческие и экономические проблемы, которые все еще не позволяют достичь преимуществ, связанных с повторным использованием.

На начальной стадии перехода к объектно-ориентированным методам необходимо задействовать самых опытных специалистов и использовать самые лучшие средства. Выделите реальный бюджет, распределите административные обязанности и выберите небольшие, но вместе с тем важные для коммерческой деятельности приложения. Возможности по использованию объектно-ориентированных методов чрезвычайно широки.

2.6. Дополнительная литература

Самое исчерпывающее описание преимуществ объектно-ориентированного подхода в целом и мотивов его принятия в профессиональной сфере можно найти в [750]. В данной работе предоставляется также обширный материал, связанный с конкретными ситуациями применения этой технологии. Дополнительное исследование примеров использования объектного подхода приведено также в [362] и трудах некоторых конференций, посвященных объектно-ориентированной технологии, а именно *TOOLS*, *OOPSLA*, *Object Expo* и *Object World*.

В замечательных книгах [548, 549] содержится полное описание причин перехода на объектно-ориентированную технологию и преимуществ ее использования в программировании и проектировании, хотя при разъяснении понятий основное внимание уделяется языку Eiffel, который был разработан самим Мейером. При написании данной главы очень многое было позаимствовано именно из этих книг, так что студентам, занимающимся объектно-ориентированным программированием, стоит прочитать обе книги. В работе [198] также дается вполне исчерпывающая оценка преимуществ объектно-ориентированного программирования и большое внимание уделяется повторному использованию программных элементов. В большей части книг, посвященных объектно-ориентированному анализу и проектированию, рассматриваются преимущества объектной технологии общего характера, однако их список слишком велик, чтобы быть приведенным в данной книге.

В [60] приведены критические замечания о возможности объектной технологии гарантировать широкое повторное использование и предлагается альтернативный подход к решению этой проблемы. В [745] критикуется ограниченность традиционного взгляда на объектную технологию и высказывается предположение о том, что все проблемы связаны с тем, что классы являются слишком малыми для повторного использования, и что наследование используется не надлежащим образом.

2.7. Упражнения

1. Каково главное отличие объектно-ориентированных систем от традиционных?

   а) возможность повторного использования объектно-ориентированных систем значительно выше

   б) в объектно-ориентированных системах применяется наследование

   в) объектно-ориентированные системы более современны

   г) изменения структур данных локализованы

   д) объектно-ориентированные системы позволяют снизить стоимость разработки

2. Сформулируйте в общих чертах восемь преимуществ объектной технологии.

3. В общих чертах сформулируйте четыре заблуждения, связанных с объектно-ориентированным подходом.

4. Опишите один из известных вам объектно-ориентированных проектов и подробно проанализируйте все преимущества, полученные от использования объектной технологии, и проблемы, с которыми пришлось столкнуться. В качестве альтернативы опишите обычный проект и продумайте, каким образом объектная технология могла бы облегчить и/или осложнить его выполнение.

3

Объектно-ориентированные и объектные языки программирования

Язык — это одеяние мысли.

Д-р Джонсон.
Жизнеописания английских поэтов

В главе 1 достаточно неформально мы уже ознакомились с важнейшими понятиями, лежащими в основе объектно-ориентированного программирования и его методов. Наряду с рассмотрением языков программирования в данной главе уделяется немного больше внимания специальной терминологии. Данная книга посвящена не только объектно-ориентированным языкам программирования, но их роль в развитии объектно-ориентированных идей так велика, что необходимо предоставить хотя бы краткий обзор. Однако мы затронем, в основном, перспективные направления в области разработки программного обеспечения, и языки, представленные в этой главе, будут рассматриваться с точки зрения их возможного принятия на коммерческой основе.

На практике существует свыше 100 объектно-ориентированных и объектных языков программирования. Конечно, потребовалось бы очень много места, чтобы хотя бы перечислить все из них. В распоряжении читателей, желающих изучить объектно-ориентированные языки программирования, имеется множество хороших книг, ориентированных на такое обучение; эти книги приведены в завершающем разделе данной главы.

3.1. Объектно-ориентированные языки программирования

Пожалуй, невозможно дать полное и точное определение объектно-ориентированного языка программирования. Однако из главы 1 нам известно, что "объектно-ориентированный" означает основанный на классах язык, в котором реализовано наследование и рекурсия (обращение функции к самой себе — *примеч. пер.*). В этом разделе мы рассматриваем языки, которые соответствуют такому определению и почти всеми специалистами воспринимаются как объектно-ориентированные. Схоластические доводы по поводу того, являются ли эти языки в полном смысле слова объектно-ориентированными, — это совершенно бесплодные суждения, особенно в свете того обстоятельства, что здесь не существует формальной теории, которая есть у логического и функционального программирования. До тех пор, пока не появится такая теория, не может существовать никакого объектно-ориентированного эквивалента двенадцати правилам Кодда для реляционных баз данных [176].

3.1.1. Язык Simula

Описываемый здесь язык Simula (ранее называемый Simula 67) был, вероятно, первым языком, в котором появились понятия классов и наследования. Он возник в 1967 году (см. работы [207, 208]) на базе более ранних работ над специальным языком Simula 1, предназначенным для моделирования дискретных событий, и это повлияло на многие его свойства. В свою очередь язык Simula 1 и все его потомки испытывали сильное влияние со стороны языка Algol, для которого характерно понятие программных блоков.

Своим появлением язык Simula обязан работе над моделированием дискретных событий и исследованием операций, датируемой 1949 годом, когда Найгард (Nygaard) работал над проектом, связанным с ядерными реакторами. При рассмотрении подобных проблем приходится сталкиваться с дифференциальными уравнениями, которые слишком сложны и не допускают даже численного решения. Поэтому с самого начала пришлось разрабатывать альтернативные подходы. Идея состояла в том, чтобы смоделировать скачкообразно эволюционирующий (изменяющий свое состояние) мир в соответствии с некоторыми временны*ми* интервалами. При использовании таких моделей, которые могут быть неоднократно "пропущены через компьютер", последствия задания различных параметров могут быть обнаружены экспериментально, а не просчитаны. Это можно сравнить с использованием электронных таблиц для проведения бизнес-анализа "что..., если...". Однако языки, применяемые в 1960-х годах, не были в достаточной степени пригодны для моделирования. Тогда как обычные языки, например язык FORTRAN, пытались описать процесс, происходящий в компьютере; язык Simula с его абстракциями пробовал описать процессы, происходящие в реальном мире.

Проблема конструктивного моделирования предполагала наличие объектов, которые описывали бы нейтроны и абсорбирующие стержни в ядерном реакторе. Рассмотрим более простой для понимания пример: моделирование движения транспорта, проходящего через сложное пересечение магистралей. Такое моделирование включает следующие параметры: интенсивность случайного входящего потока, интенсивность потока насыщения, регулировку светофора и организацию очередей. Появление и перемещение транспортного средства отслеживается системой по мере прохождения дискретных временны*х* интервалов, что скорее похоже на замедленную киносъемку; и на каждой временно*й* стадии поведение может быть исследовано с учетом значений длины очереди, времени задержки, блокировки перехода и

аналогичных параметров. Возможно также наблюдение глобальных явлений, например устойчивости и приближения к устойчивому состоянию. Однажды сконструированная система может быть протестирована в условиях задания разных значений различным переменным: объема, связывания сигналов или периода цикла.

Язык Simula, который все еще имеет немало активных пользователей, — это язык программирования со встроенными понятиями классов и иерархического наследования. Множественное наследование не поддерживается. Сокрытие информации достигается при помощи свойства "защищенности", которое, в сущности, запрещает дальнейшее наследование члена класса. Полиморфизм поддерживается в форме перегрузки. Проверка типов может выполняться статически либо на стадии компиляции (в целях повышения эффективности), либо на стадии выполнения, если член объявлен "виртуальным". Время от времени в процессе выполнения система должна осуществлять сборку мусора, т.е. она должна очищать память, отведенную под экземпляры, которые были заданы, но больше не подлежат использованию. В языке Simula, как и в большинстве других объектно-ориентированных языков программирования, это свойство снижает эффективность работы программы на стадии выполнения, но мы уже видели, что основное влияние на развитие языка Simula оказали проблемы моделирования, для которых производительность не играет существенной роли. Язык Simula обладает библиотекой специальных классов, которые содержат основные примитивы, необходимые для моделирования дискретных событий; почти во всех остальных объектно-ориентированных языках такие примитивы должны писать программисты. С принципами моделирования, характерными для языка Simula, связано также существование набора уникальных свойств, предназначенных для поддержки приложений, придающих большое значение взаимной совместимости; "сопрограммы" могут вызываться и продолжать функционировать, не прерывая выполнения основного процесса до тех пор, пока не потребуются результаты работы сопрограммы.

В языке Simula основное внимание уделяется не данным, а процессам, однако этот язык содержит в себе множество идей, которые были реализованы его преемниками, — это касается, главным образом, классов, экземпляров и наследования. Идея абстракции неявно уже присутствовала, но такие понятия, как, например, передача сообщений, ждали появления на сцене языка Smalltalk (его основные свойства приведены в табл. 3.1). Специалисты, работавшие над языком Simula, стали авторами нового языка — BETA [461], который не только является объектно-ориентированным, но и включает элементы функционального и основанного на ограничениях программирования; он описан далее в разделе 3.6.1. Та же группа разработчиков создала строго типизованный объектно-ориентированный язык формальных описаний, который получил название ABEL [206].

Таблица 3.1. Свойства языка Simula

Проверка типов/связывание	Позднее или раннее
Полиморфизм	Да
Сокрытие информации	Да
Параллелизм	Да
Наследование	Да
Множественное наследование	Нет

Окончание табл 3.1

Сборка мусора	Да
Перманентность	Нет
Универсальность	Нет
Объектные библиотеки	Моделирование

3.1.2. Язык Smalltalk и его диалекты

В настоящее время язык Smalltalk широко признан как абсолютное воплощение объектно-ориентированного идеала. До слияния компаний-разработчиков Smalltalk существовал в виде двух диалектов: Smalltalk 80 и Smalltalk V (это не римская цифра V, а буква "ви", т.е. виртуальный). Язык Smalltalk появился в компании Xerox PARC как результат работы Алана Кея (Alan Kay), Адели Голдберг (Adele Goldberg), Дэниэля Ингаллса (Daniel Ingalls) и других разработчиков, которая длилась на протяжении 1970-х годов. По сути, Smalltalk — это не просто язык, а целая среда программирования, содержащая редакторы, броузеры иерархий классов и многие другие средства языков программирования четвертого поколения. Именно в языке Smalltalk было явно введено понятие передачи сообщений как основного и единственного способа связи между объектами. Испытывая влияние со стороны языка Simula, разработчики языка Smalltalk довели понятие объектов до крайности и объявили, что объектами должно быть абсолютно все, включая сами классы. В языке Smalltalk даже простое число является объектом. В силу такого подхода программисту приходится иметь дело с классами классов — метаклассами, и некоторые авторитетные специалисты утверждают, что достаточно загадочное понятие метакласса в языке Smalltalk приводит к тому, что этот язык становится сложным для понимания, по крайней мере для начинающих. Кроме того, рассматривать числа как объекты очень удобно с точки зрения совместимости, но это не всегда эффективно и своей необычностью сбивает с толку большинство программистов, работавших в других языках. Язык Smalltalk не является языком со строгой типизацией, так что ошибки, возникающие, например, во время получения объектом сообщения, которое он не должен получать, обрабатываются полностью на стадии выполнения. Он нуждается также в сборке мусора, и все связывание производится только на стадии выполнения. Все это означает, что с точки зрения времени выполнения язык Smalltalk никоим образом нельзя считать быстрым. Это компенсируется тем, что он представляет собой превосходную среду разработки, особенно для модельного представления, и является, возможно, наилучшим из известных инструментов по созданию оконно-графических интерфейсов.

Таким образом, благодаря работе над языком Smalltalk, в программирование не только пришло понятие *объектно-ориентированный*, но и появились идеи, лежащие в основе специальных графических пользовательских интерфейсов, которые можно обнаружить во многих современных системах, начиная с Apple Lisa. Возможно, именно простотой создания таких интерфейсов в языке Smalltalk объясняется преобладание на начальном этапе графических интерфейсов, а не других приложений в сфере объектно-ориентированного программирования, и существование лишь немногих объектных библиотек, предназначенных для других приложений.

Согласно оценкам, объем работ над проектом Lisa составил свыше 200 человеко-лет. Очевидно, что такая компания, как Apple, не может позволить себе такие затраты при каждой

реализации новой машины. Поэтому выгоды от повторного использования программного обеспечения становятся для таких приложений не просто преимуществом, а необходимостью.

Метакласс представляет собой важное и имеющее решающее значение понятие, которое было введено в языке Smalltalk. Говоря на обычном языке, метаклассы — это просто классы классов, которые не могут иметь экземпляров и поэтому являются некоторыми абстрактными понятиями, от которых могут быть унаследованы более конкретные понятия. В языке Smalltalk упомянутое выше понятие получило название абстрактного класса, а метакласс определяется как шаблон для создания класса, т.е. нечто, что способно создавать классы. Это безусловно объясняется влиянием языка Lisp, а также позволяет описывать язык внутри языка и действительно расширять и изменять структуру языка. В языке Smalltalk каждый класс является единственным экземпляром соответствующего метакласса и метакласс создается одновременно с классом. Каждый метакласс является экземпляром класса `Metaclass`, а его подкласс — класса `Class`. Методы класса на самом деле хранятся в метаклассе. Этот язык не имеет встроенных типов (только тип `object`), и некоторые специалисты рассматривают классы как типы, определяемые пользователем. Это еще одно свидетельство влияния на Smalltalk языка Lisp и функционального программирования.

Язык Smalltalk не обладает встроенной структурой управления; любая такая структура должна создаваться на основе передачи сообщений. Например, если выражение заключается в квадратные скобки — так называемый "блок" — его вычисление откладывается. Как правило, управляющие структуры реализованы как сообщения для объектов, которым в качестве аргументов передаются блоки.

Язык Smalltalk поддерживает только простое (одиночное) наследование, поэтому приложения, нуждающиеся во множественном наследовании, с целью преодоления этого ограничения должны быть спроектированы, вероятно, несколько искусственным образом. В [544] утверждается, что при этом часто происходит значительное дублирование кода и сильно усложняется динамическое связывание.

Управление изменениями играет важную роль в классической среде языка Smalltalk, поскольку программисты, в принципе, могут изменить работу любого своего коллеги. Броузер среды Smalltalk поддерживает хорошую связь между членами команды, а также обеспечивает другие средства управления. Главный недостаток языка Smalltalk — отсутствие хоть какого-нибудь понятия перманентных объектов. Еще одним недостатком является однопользовательский характер. Среда разработки ENVY явилась в некоторой степени ответом на этот недостаток. ENVY предоставляет репозитарий (хранилище) совместно используемого кода, который содержит реализации, версии и выпуски классов, методов и связанных с ними перманентных объектов. Таким образом становится возможным управление конфигурацией в распределенной среде, что позволяет выполнять проекты, над которыми работает множество программистов. До появления таких продуктов использовались альтернативные системы, которые были ориентированы на работу с файлами и осуществляли проверку регистрации, но были довольно неуклюжими. Впоследствии разработчики среды ENVY разработали новый компилятор IBM Visual Age для среды Smalltalk, который строится во многом на все тех же идеях, а также включает мощную среду визуального программирования.

Хотя язык Smalltalk предоставляет в распоряжение программистов исчерпывающую рабочую среду, но все же такие свойства, как сборка мусора, динамическое связывание и обширный графический пользовательский интерфейс, подразумевают жесткие требования к аппаратным средствам. Кроме того, язык Smalltalk — это самостоятельная среда, которая (до появления CORBA) не могла легко взаимодействовать с другими языками и не позволяла

интегрировать существующий код. Эти два основных фактора препятствуют ее широкому распространению.

К преимуществам языка Smalltalk можно отнести

- концептуальную однородность, выраженную в том, что любой элемент считается объектом;
- замечательную среду выполнения, включающую символьные отладчики, броузеры и полный доступ к иерархии классов;
- динамическое связывание, поддерживающее полиморфизм, а следовательно, и значительную гибкость;
- автоматическое управление памятью;
- легкость понимания и управления всем проектом в целом по сравнению с обычными языками;
- тот факт, что, работая на языке Smalltalk, трудно не написать код в объектно-ориентированном стиле, что делает язык превосходным средством обучения. Он может быть использован для обучения ключевым понятиям с той целью, чтобы у программистов вошло в привычку писать код, придерживаясь объектно-ориентированного стиля даже при работе с такими языками, как язык C++, где стиль не является строго предопределенным.

Теперь рассмотрим недостатки Smalltalk:

- недостаточная эффективность и потребность в значительных размерах памяти;
- нет поддержки перманентных объектов;
- ошибки при передаче сообщений могут быть обнаружены только на стадии выполнения.

После появления языка Smalltalk было сделано множество попыток создать объектно-ориентированный язык (с риском разрушения чистоты языка Smalltalk и совместимости), способный решить проблемы эффективности и сохранить основные преимущества объектно-ориентированного программирования. (Основные свойства Smalltalk представлены в табл. 3.2.) Эти попытки можно отнести к различным категориям, и оцениваться они должны по разным критериям. Во-первых, существует различие между языками, акцентирующими все внимание на абстракции в ущерб наследованию и наоборот. Во-вторых, существует разница между языками, следующими традиции искусственного интеллекта, и языками, которые своим происхождением обязаны продуманным решениям специалистов по компьютерным языкам. И наконец, существует три направления, которых придерживаются специалисты по логическому, функциональному и другим видам программирования. Большая часть лагеря специалистов, занимающихся традиционным программированием, взяла курс на расширение существующих языков, дополняя их объектно-ориентированными свойствами. Именно на этом мы сейчас кратко остановимся.

Таблица 3.2. Свойства языка Smalltalk

Проверка типов/связывание	Позднее
Полиморфизм	Да
Сокрытие информации	Да
Параллелизм	Слабый
Наследование	Да
Множественное наследование	Нет
Сборка мусора	Да
Перманентность	Нет
Универсальность	Нет
Объектные библиотеки	Несколько

3.1.3. Расширения языка C

Один из популярных подходов к эффективному объектно-ориентированному программированию состоит в расширении существующего языка посредством дополнения его объектно-ориентированными свойствами. Язык C может служить примером эффективного базового языка, расширенного таким образом. По сути, имеются два способа, определяющие, как это можно сделать; соответствующими примерами могут служить языки Objective-C и C++. Язык Objective-C [197, 198] расширяет возможности простого языка C, включая один дополнительный тип данных в виде структуры языка C в рамках библиотеки функций, имеющей, по сути, функциональность языка Smalltalk. С другой стороны, язык C++, разработанный в компании AT&T Страуструпом [733, 736], действительно изменяет компилятор C и дополняет синтаксис новыми примитивными типами данных — классами.

Язык Objective-C

Objective-C — это язык, созданный на базе обычного языка C, но имеющий библиотеку, которая обеспечивает значительную часть функциональности, присущей элементам языка Smalltalk. Он содержит новый тип данных (идентификатор объекта) и новую операцию — передачу сообщений и представляет собой просто расширенный набор команд языка C. Первоначально язык Objective-C функционировал как препроцессор, который преобразовывал команды в представление, приемлемое для обычного компилятора C. Сейчас во многих средах он действует как самостоятельный компилятор.

Терминология Кокса (Cox) отличается от терминологии, принятой в языке Smalltalk. Хранением информации управляют объекты-"фабрики", которые представляют классы и создаются на стадии компиляции. Все объекты-фабрики содержат метод `new`, который создает экземпляры соответствующих классов на стадии выполнения, реагируя на сообщения следующего вида.

```
thisObject = [Object new]
```

В этих объектах инкапсулированы методы и переменные состояния, унаследованные экземплярами. Экземпляры инкапсулируют значения состояния и имеют уникальный идентификатор. Унаследованный член может быть перекрыт, если экземпляр или подкласс будет содержать переменную или метод с таким же именем. Множественное наследование классов не поддерживается, но может быть реализовано программистом. Однако в языке Objective-C появилась идея "протоколов", т.е. интерфейсов, которые могут наследоваться классами независимо от реализации этих интерфейсов. Разработчики языка Java реализовали эту идею, которая известна теперь под названием интерфейса `interface`. Таким способом поддерживается множественное наследование интерфейсов.

В языке Objective-C (табл. 3.3) предусмотрены некоторые встроенные классы, например классы объектов, массивов, коллекций, последовательностей, множеств и т.д. Доступны также библиотеки классов. С точки зрения стандарта этот язык был связан с машиной NeXT и использовался для написания самых длинных из когда-либо создававшихся объектно-ориентированных программ. В настоящее время среда OpenStep функционирует на базе нескольких машин и располагает огромной библиотекой классов, которая предлагает эффективный доступ программистам, пишущим на языке Objective-C. Вне этой среды язык Objective-C не получил широкого распространения.

Таблица 3.3. Свойства языка Objective-C

Поверка типов/связывание	Позднее (по умолчанию) или раннее
Полиморфизм	Да
Сокрытие информации	Да
Параллелизм	Слабый
Наследование	Да
Множественное наследование	Только для интерфейсов
Сборка мусора	Да
Перманентность	Нет
Универсальность	Нет
Объектные библиотеки	Несколько + среда OpenStep

Язык C++

Как известно, язык C++ представляет собой скорее не новый язык, а совместимое снизу-вверх расширение непосредственно языка C. Принципиальное изменение заключается в появлении нового реального примитивного типа данных — класса. Этот язык не содержит типов данных высокого уровня и примитивов, а, подобно языку C, обеспечивает расширения при помощи библиотек. Таким образом, новые типы задаются непосредственно внутри самого языка. На этот язык оказали влияние, главным образом, два языка: C (и его замечательный предшественник — язык BCPL) и Simula. При создании язык C++ задумывался как мобильный и эффективный язык. В действительности причиной разработки языка C++ послужил определенный недостаток языка Simula 67 в вопросах эффективности. Кроме

того, в языке C++ особое внимание уделяется возможности поддержки значительной части существующего кода, написанного на языке C. Кроме того, программист может нарушить защиту данных и правила типизации, в результате чего доступ к данным может оказаться неправильным, хотя сделать это случайно намного сложнее, чем в языке C. Язык C++ представляет собой компромиссное решение между объектно-ориентированным идеалом и прагматичным подходом.

Язык C++ поддерживает абстракцию, наследование, рекурсию и динамическое связывание. Поддерживается также и статический, и динамический контроль типов. Система типов, в отличие от языка Smalltalk, организована не в виде однокорневой древовидной структуры, а представляет собой коллекцию относительно небольших деревьев. Поэтому для C++ характерна тенденция поддерживать широкие и плоские структуры классов, чем он и отличается от языка Smalltalk, способствующего созданию узких и глубоких структур классов, поскольку здесь любой класс является подклассом класса `Object`. Подобное наблюдение может быть полезно при определении подходящей области применения.

При объявлении классов некоторые их части могут задаваться как открытые, защищенные и закрытые. Внутри класса закрытые данные и методы доступны только для собственных методов объекта, тогда как к открытым данным и методам может обращаться любой другой объект или функция. В языке C++ методы называются функциями-членами. Передача сообщения соответствует обращению к функции. Данные хранятся в переменных экземпляра (согласно терминологии языка C++ — закрытых переменных-членах). Определенным образом — посредством выделения памяти при загрузке программы — данные могут сохраняться и на уровне классов. Таким образом эти данные фиксируются для всех экземпляров класса.

Одно оригинальное свойство языка C++ связано с возможностью предоставления некоторой функции (или классу) привилегированного доступа к закрытым частям нескольких классов, членом которых эта функция (или класс) не является. С этой целью такая функция объявляется "другом" данных классов. Это свойство может быть использовано для упрощения описания операции над двумя типами. Предположим, нам нужно задать умножение векторов на матрицы, тогда, определяя операцию умножения в качестве друга этих классов, мы открываем ей доступ к их реализациям. Функции-друзья должны содержать один дополнительный аргумент, указывающий объект, участвующий в операции. Благодаря друзьям поддерживается возможность вызова функций языка C++ из программ, написанных на обычном языке C. Это свойство требует крайне осторожного обращения ввиду возможного нарушения инкапсуляции.

Так же, как и в языке Smalltalk, разные методы могут иметь одинаковые имена. Таким образом, умножение целых чисел или матриц, например, может быть представлено одним и тем же символом. Возвращаясь к главе 1, читатель вспомнит, что такое свойство называется *перегрузкой операции* и является некоторой особой формой полиморфизма.

Наследование реализуется посредством объявления подклассов. В языке C++ подклассы называются *производными классами*. Производные классы наследуют в качестве открытых все открытые свойства базового класса, причем в том виде, в котором они были заданы. Возможно существование иерархий, поскольку сами по себе базовые классы могут быть производными. Унаследованные методы могут быть перекрыты. Множественное наследование поддерживается полностью, а весь механизм полной обработки исключительных ситуаций представлен набором команд `try/throw/catch`.

По умолчанию связывание, как и в обычном языке, производится во время компиляции, и переменная считается либо глобальной, либо видимой в пределах блока. С другой стороны, объект может создаваться в свободной области памяти, тогда его определением и удалением

управляет программа. При использовании этих технологий типы аргументов функции должны быть известны на стадии компиляции. Динамическое связывание реализуется при помощи *виртуальных функций*, а интерфейсы (или отложенное наследование) — при помощи *чисто виртуальных функций*, которые описываются, но не реализуются в базовом классе и могут быть переопределены в производных классах. Это значит, что типы аргументов могут определяться в момент вызова функции.

В языке C++ не предусмотрена автоматическая сборка мусора. Подобное свойство должно реализовываться программистом или предоставляться средой в виде соответствующих библиотек кода. Однако в настоящее время существуют некоторые сборщики мусора, которые добавляются в качестве расширения языка (см. `ftp:/parcftp.xerox.com/pub/gc/gc.html`). Последними в язык C++ были добавлены шаблоны. С их помощью поддерживается неограниченная универсальность; однако шаблоны были подвергнуты критике ввиду того, что они приводят к раздуванию кода и замедлению компиляции. Существует стандартная библиотека шаблонов (STL), которая может использоваться любым компилятором [298]. Макрос `assert` позволяет производить некоторую формальную проверку кода во время отладки, дополняя проверку, выполняемую с помощью команд `try/throw/catch`. Однако язык C++ не обладает такими мощными средствами формальных утверждений, как язык Eiffel.

Язык C++ можно использовать в качестве языка объектно-ориентированного программирования, однако все (или некоторые) его объектно-ориентированные свойства можно игнорировать или неправильно использовать. Весьма отрезвляющие статистические данные свидетельствуют о том, что среди многочисленных программ, написанных на C++ в компании AT&T в 1989 году, менее чем одна из десяти содержала хотя бы одно определение класса. Вряд ли с тех пор улучшилось понимание объектно-ориентированного подхода. Вероятно, программистам, пишущим на языке C++ и прежде писавшим на языке C, следует программировать на языках Smalltalk, Java или Eiffel (возможно, моделируя проектируемые реализации на C++), где они могут забыть некоторые свои более глубоко укоренившиеся привычки и вынуждены будут думать категориями объектов. Это говорит также о необходимости создания согласованного объектно-ориентированного проектного решения и методов анализа.

В настоящее время используется несколько различных версий языка C++, некоторые из них являются открытыми и общедоступными (например, компилятор Gnu) на базе самых разнообразных машин.

Как я уже отмечал, язык C++ — это компромисс, однако разумный компромисс. Этот язык позволяет при необходимости осуществлять низкоуровневое управление аппаратными средствами, а также воспользоваться преимуществами объектно-ориентированного программирования. Самое плохое, что можно сказать о языке C++, — это, во-первых, его значительная сложность при правильном применении, а во-вторых, продолжающееся использование его многими программистами просто как "улучшенного" языка C. В некоторых случаях он действительно может рассматриваться скорее как язык системного программирования, а не язык разработки приложений: машинно-независимый ассемблер! Несмотря на все эти замечания язык C++ является одним из наиболее приемлемых объектно-ориентированных языков программирования для коммерческих приложений, предназначенных для рабочих станций и персональных компьютеров, и скорее всего оставался бы таковым на протяжении обозримого будущего, если бы не появился язык Java. Известно огромное количество проектов, содержащих программы, написанные на языке C++. В их разработке задействованы сотни специалистов, тысячи классов и миллионы строк кода. Продолжительный успех этого языка объясняется хорошими рабочими свойствами (которые не позволяли

пробиться первым приложениям, написанным на Java) и дальнейшим развитием высококачественных библиотек компонентов.

Преимущества языка C++ таковы:

- возможно выполнение любых действий на любом уровне операционной системы;
- вероятно, язык C++ является самым быстрым из существующих в настоящее время объектно-ориентированных языков;
- рынок программистов, пишущих на языке C++, достаточно велик.

Недостатки C++ заключаются в следующем:

- отсутствие автоматического управления памятью (сборки мусора) и широко распространенное использование указателей приводят к тому, что гарантии безопасности на стадии выполнения достигаются с большим трудом;
- очень сложно изучить этот язык и стать *хорошим* программистом, пишущим на языке C++.

В [737] особенно сильные стороны языка C++ (табл. 3.4) лаконично сформулированы так: это язык "для приложений, которые содержат компоненты системного программирования, для систем, предъявляющих высокие требования ко времени и пространству, и систем, которые затрагивают некоторые технические области приложения. В таких системах более простые, менее эффективные и более специализированные языки становятся источником неприятностей".

Таблица 3.4. Свойства языка C++

Поверка типов/связывание	Позднее или раннее (по умолчанию)
Полиморфизм	Да
Сокрытие информации	Да
Параллелизм	Слабый
Наследование	Да
Множественное наследование	Да
Сборка мусора	Нет
Перманентность	Нет
Универсальность	Да — шаблоны
Объектные библиотеки	Весьма многочисленные

3.1.4. Язык Eiffel

Созданный Мейером [545] язык Eiffel представляет собой разработанный для определенной цели объектно-ориентированный язык, который сознательно пытается решить проблемы корректности, робастности, переносимости и эффективности. На его разработку значительно

повлияли языки COBOL, Ada и Simula. В отличие от языков Smalltalk и Objective-C, в языке Eiffel классы не являются объектами, поскольку здесь они идентифицируются как модули, что позволяет осуществлять статический контроль типов в целях устранения ошибок на стадии выполнения и повышения эффективности.

Класс описывает реализацию абстрактного типа данных. Он определяется на стадии компиляции и может создавать экземпляры данного типа (т.е. объекты) на стадии выполнения. Терминология языка Eiffel отличается от терминологии, принятой в других языках, — здесь методы называются *подпрограммами*. В классах инкапсулированы и подпрограммы, и атрибуты, которые принято называть *свойствами*. Свойства могут быть закрытыми или общедоступными (экспортированными), точно так же как в языках Java, C++ или Ada.

Важной характеристикой языка Eiffel является его способность описывать формальные свойства, которым должны подчиняться операции объектов, посредством формулировки *утверждений*. При помощи утверждений могут задаваться предусловия и постусловия для методов или инвариантов класса. Предусловия вынуждают делать проверку при каждом обращении к методу, постусловия гарантируют правильность завершения метода или возврата значений, а инварианты всегда остаются правильными, если однажды был создан объект или вызван метод. Эти свойства содействуют достижению цели — обеспечить корректность — и отражают взгляды Мейера на разработку программного обеспечения и формальные методы. Свойства подобного рода могут быть добавлены в язык Smalltalk без особых усилий, что продемонстрировал проект Fresco в Манчестерском Университете [790]. Язык Eiffel в равной степени может считаться как языком проектирования, так и языком программирования. Проектное решение реализуется как набор классов высокого уровня и методов, заполняемых позднее. При этом уменьшается риск появления ошибок во время перевода формальных понятий проектного решения в код. Этот язык невелик и прост в изучении, но поставляется вместе с обширной средой программирования и отличной библиотекой классов. На него сильно повлияли почти все современные объектно-ориентированные методы, особенно это касается утверждений.

В целях эффективности и переносимости приложения компилируются в язык C. С точки зрения эффективности чрезвычайно важно, что сборка мусора не является обязательной. При включении сборки мусора ненужные данные удаляются постепенно, в соответствии с оригинальным алгоритмом Eiffel.

Утверждения позволяют вводить в язык Eiffel средства обработки исключительных ситуаций. Нарушение может привести к сбою сообщения или вызвать "спасательную" процедуру, содержащуюся в методе-нарушителе. Утверждения могут рассматриваться как формальное соглашение между сервером и его клиентами. Для повышения эффективности во время окончательной компиляции механизм утверждений может быть отключен.

Как и в языке Ada, здесь существует понятие параметризованных (родовых) классов, т.е. классов, которые имеют параметры, задающие типы; при этом поддерживается ограниченная форма универсальности и отсутствует снижение производительности, характерное для шаблонов языка C++. Множественное наследование поддерживается, но конфликтные ситуации, связанные с наследованием, не обрабатываются, поскольку в потомках атрибуты или методы, унаследованные от двух родителей, должны быть переименованы. Заметим, что до сих пор во всех рассматриваемых языках значения атрибутов не наследуются, наследуется только способность иметь значение. Включение полиморфизма поддерживается ввиду возможности контроля типов с целью допущения более специфичных (но не более общих) объектов. В языке Eiffel допускается перекрытие унаследованных методов.

В настоящее время в язык Eiffel включен простой механизм параллелизма, который основан на добавлении единственного ключевого слова — `separate`. Еще одним бесценным свойством этой среды является ее технология Melting Ice (дословно — таяние льда — *примеч. пер.*). Это значит, что исходный код может рассматриваться как жидкость, которая "замораживается" при компиляции. В большинстве языков модули должны быть полностью разморожены перед тем, как можно будет произвести их изменение, а затем заморожены снова и привязаны к другим модулям. В языке Eiffel необходимо разморозить только те строки кода, которые должны быть изменены. Затем их можно тестировать как интерпретируемый код, функционирующий параллельно с другими, все еще замороженными элементами системы. Разработчики языка Eiffel в значительной степени полагаются на использование множественного наследования, способствующего созданию действительно многократно используемых компонентов [547].

Хотя появление языка Eiffel опередило свое время и обеспечило немаловажное развитие понятий объектно-ориентированного программирования, первые его версии, рассматриваемые как система, предназначенная для коммерческих разработок, получили довольно плохие отзывы в прессе. На конференции ECOOP'89 [185] было высказано мнение, что язык Eiffel не является безопасным с точки зрения типов. В [475] сообщается, что поставщик языков программирования четвертого поколения — Cognos — прекратил разработку проектов на основе языка Eiffel ввиду проблем с производительностью, присущих ранним версиям языка Eiffel, а также некоторой нехватки сервисных средств (броузеров и т.д.) и, что более важно, из-за ошибок в управлении незнакомой технологией. Лизерс (Leathers) обрисовывает некоторые важные моменты: недостаток квалифицированного персонала, плохое понимание технологии и невозможность применять обычные методы управления проектом. Самый важный вывод, который можно сделать на основании этого, заключается в том, что создание прототипов имеет огромное значение. Создание прототипа на первой стадии развития проекта позволило бы принимать более рациональные решения, которые должны затрагивать также и окончательную разработку. В главе 9 мы подробно рассмотрим вопрос прототипирования и управления им. Совсем недавно Чикагская торговая палата успешно использовала язык Eiffel в своих базовых торговых системах.

Самые последние разработки включают новые, в значительной степени усовершенствованные версии исходного компилятора. В настоящее время этот язык общедоступен. Стали появляться некоторые очень хорошие компиляторы других фирм. Многие считают язык Eiffel (табл. 3.5) самым лучшим из существующих объектно-ориентированных языков, но все же, несмотря на многочисленные успешные отдельные проекты, он не получил широкого признания. Сейчас самая серьезная угроза для него исходит от популярного языка Java, хотя язык Eiffel имеет некоторые свои преимущества. Механизм *агентов* языка Eiffel реализует идеи, позаимствованные из функциональных языков (например, языков Lisp и Miranda), наделяя, таким образом, разработчика прототипа или специалиста по экстремальному программированию еще большей гибкостью. Заметим, что подобное использование слова "агент" не имеет ничего общего с интеллектуальными или мобильными агентами, рассматриваемыми в главах 8 и 10.

Таблица 3.5. Свойства языка Eiffel

Поверка типов/связывание	Раннее (по умолчанию)
Полиморфизм	Да
Сокрытие информации	Да
Параллелизм	Да
Наследование	Да
Множественное наследование	Да
Сборка мусора	Да
Перманентность	Частичная поддержка
Универсальность	Да
Библиотеки объектов	Поставляются компанией ISE Inc.

3.1.5. Язык Java

Когда появился язык Java, он завоевал рынок штурмом. И это объяснялось рядом причин.

- Этот язык предусматривал работу во всемирной сети, поддерживая безопасность и параллелизм. Небольшие написанные на языке Java программы могли быть предоставлены в виде *аплетов* для работы в броузере, например, Internet Explorer или Netscape Navigator.
- Кроме того, на нем могут быть написаны абсолютно самостоятельные приложения.
- Синтаксис этого языка похож на синтаксис C++, однако Java является чисто объектно-ориентированным языком. Он обеспечивает автоматическое управление памятью, не использует указатели и поэтому является намного более безопасным.
- Он появился как всеобщее достояние, однако поддерживался мощным производителем — компанией Sun.

Исходный код Java компилируется в псевдокод (p-код или байт-код), для интерпретации которого нужна виртуальная машина, созданная по образцу значительно более старой машины USCD-Pascal [594]. Виртуальная машина Java может работать практически на любой платформе. Это делает разработку фактически машинно-независимой и обеспечивает универсальную переносимость, однако за счет низкой производительности. Именно поэтому многие платформы вскоре обзавелись собственными компиляторами. Кроме того, существуют оперативные JIT-компиляторы (just-in-time), которые превращают байт-код в собственный код при первом запуске аплета. Впоследствии выполнение происходит намного быстрее. Сегодня новые технологии компиляции изменяются очень быстро, поэтому для выяснения этого вопроса советуем читателю обратиться к специальной литературе. Машинная независимость получает дальнейшее развитие, поскольку пакет абстрактных оконных компонентов поставляется как пакет независимых от платформы виджет-классов (виджет — элемент управления окном — *примеч. пер.*): первоначально библиотеки AWT (Java Abstract Window Toolkit), теперь — Swing.

Язык Java происходит от языка Oak, который был разработан специально для таких приборов, как микроволновые печи и кинокамеры; поэтому он должен был быть небольшим. Однако стандартный язык Java не так уж и мал. При разработке программ появляется необходимость обращаться ко многим классам, содержащимся в пакетах, поставляемых вместе с языком. Стандартные пакеты (с расширениями `.io`, `.lang` и `.util`) содержат примерно 350 методов. Пакеты обеспечивают уровень организации примерно на уровне классов. С целью практического использования языка Java программисту следует ознакомиться со многими стандартными библиотеками, входящими в состав его среды.

На Java (табл. 3.6) сильно повлиял язык Objective-C, а следовательно, и язык Smalltalk, несмотря на его чисто синтаксическое сходство с языком C++. Простые типы данных здесь не являются объектами, но по желанию они могут быть преобразованы в (неизменные) классы. Язык Java отделяет понятие класса от понятия интерфейса и пытается уменьшить опасность, связанную со множественным наследованием, допуская подобное наследование только для интерфейсов. Это может раздражать в том случае, когда нужно написать реализацию наследуемых свойств. Лично мне намного больше нравится подход, который принят в языке Eiffel, где единственное, что я должен сделать — это сообщить классу, что он является подклассом двух классов, и тогда этот класс станет таковым без единой строки кода. Существуют три версии языка Java: стандартная, корпоративная и микроверсия. Последняя, сохраняя дух языка Oak, предназначена для встроенных процессоров.

Таблица 3.6. Свойства языка Java

Поверка типов/связывание	Раннее
Полиморфизм	Да
Сокрытие информации	Да
Параллелизм	Потоки
Наследование	Для классов — одиночное
Множественное наследование	Только для интерфейсов
Сборка мусора	Да
Перманентность	Через интерфейс JDBC[1]
Универсальность	Нет
Библиотеки объектов	Компоненты Beans

Обработка исключительных ситуаций производится почти так же, как и в языке C++ — с помощью набора команд `try/throw/catch` — но намного точнее. Считается хорошим стилем, если каждый класс реализует интерфейс `Throwable` и объявляет, какие исключительные ситуации тот генерирует. Параллелизм обеспечивается посредством объявления облегченных потоков и протоколов синхронизации.

[1] *Средство организации доступа Java-приложений к базам данных в сети.* — Примеч. пер.

Компоненты Java Bean — это коллекции классов и необходимых им ресурсов. Они похожи на управляющие элементы OLE или Active X, но не могут служить контейнерами для других управляющих элементов. Они предназначены, главным образом, для предоставления специальных управляющих элементов окон в графическом пользовательском интерфейсе. Компоненты Enterprise Java Beans (EJB) — входящие в состав корпоративной версии языка — являются серверными компонентами и представляют намного более сложную технологию, которая будет рассмотрена в главе 7. Эти свойства и поддержка языка XML делают возможным использование языка Java для разработки серверов приложений.

В языке Java предусмотрена собственная технология использования брокеров объектных запросов — RMI (удаленный вызов метода). Эта технология позволяет приложениям вызывать методы других Java-приложений по сети. Начиная со второй версии язык Java содержит также брокер, удовлетворяющий условиям CORBA и независимый от языка. Брокеры объектных запросов рассматриваются в главе 4. Кроме того, в языке Java имеются классы, предназначенные для управления взаимодействием с реляционными базами данных JDBC (Java DataBase Connectivity) и для базовой графики.

Концепция безопасности первой версии Java предусматривает строгий запрет доступа аплетов к жесткому диску. Во второй версии реализована более гибкая модель на основе сертификатов.

3.1.6. Язык Object-COBOL

Комитет по стандартам Codasyl приступил к подготовке выпуска объектно-ориентированной версии языка COBOL в начале 1990-х годов, подчиняясь давлению групп пользователей, выступающих в защиту открытых систем и подчеркивающих важность многократного использования программного обеспечения по мере того, как размеры проектов становятся все больше и больше. Конечно, в то время некоторая разновидность языка Object-COBOL была крайне необходима тем пользователям, которые хотели воспользоваться преимуществами многократного использования и масштабируемости, нуждаясь в защите огромных инвестиций, вложенных в существующий код: согласно оценкам, это 70000 миллионов строк кода по всему миру. Однако конечный продукт — язык COBOL-97 — так долго ждал своего появления и стандартизации, что ко времени его появления многие из организаций уже перешли на языки C++, Smalltalk или объектно-ориентированные языки типа Visual Basic. Тем не менее язык Object-COBOL сохраняет свое значение (хотя не такое большое, как можно было ожидать) для немногочисленных крупных промышленных центров. В настоящее время используются два или три компилятора COBOL-97, самыми известными поставщиками которых являются такие компании, как Computer Associates (CA), Hitachi, IBM, завоевавшие рынок с самого начала.

Язык Object-COBOL обладает следующими свойствами:

- пользовательские типы данных, основанные на структурах записи языка COBOL;
- инкапсуляция;
- классы, выступающие в роли шаблонов для абстрактных типов данных;
- классификация (инстанцирование (создание экземпляров) при загрузке программы);
- передача сообщений, когда программа выполняет обращение при помощи команды `USING`;

- наследование с использованием команды COPY;
- полиморфизм, предусматривающий использование множественных точек входа и команды READ;
- возможность использования объектов, написанных на других языках;
- сборка мусора (при помощи команды CANCEL), которая удаляет отдельные объекты;
- совместимость с языком COBOL снизу-вверх;
- усовершенствованная поддержка прототипов.

Хотя маловероятно, что самые конкурентоспособные приложения будут написаны на языке COBOL, появление языка Object-COBOL будет приветствоваться некоторыми центрами, занимающимися традиционными разработками. Зрелость стиля программирования признается тогда, когда она применима к земному, а не эзотерическому.

3.2. Другие языки, обладающие объектно-ориентированными свойствами

Некоторые языки хотя и не демонстрируют полную приверженность объектно-ориентированному подходу, характерную для языка Smalltalk, но все же обладают объектно-ориентированными свойствами. Вообще говоря, они акцентируют свое внимание либо на абстракции, либо на наследовании. Такие языки, как Ada 83, Modula-2 и, возможно, даже язык Object Pascal, попадают в первую категорию. Некоторые продукты, ведущие свое происхождение от работ в области искусственного интеллекта, относятся ко второй категории, и в этом случае мы часто слышим термин *фреймовый* язык. Эта категория языков будет рассмотрена в разделе 3.4 данной главы.

Такие языки, как Ada 83 и, в меньшей степени, Modula-2, могут служить примерами языков, ключевыми понятиями которых являются абстракция данных и преимущества многократного использования. Как было сказано в главе 1, такие языки называются объектными, т.е. основанными на объектах.

В языке Ada 83 и подобных ему объектных языках пакеты не являются объектами первого класса в том смысле, что они не могут быть переданы в качестве параметров. Это значит, что язык Ada поддерживает абстракцию данных не полностью. Тем не менее этот язык действительно поддерживает перегрузку операций и предоставляет возможность выполнения сборки мусора по желанию пользователя.

В языке Ada 83 не предусмотрена непосредственная поддержка наследования, но некоторая ограниченная форма наследования может быть реализована с помощью родовых пакетов. Хотя допускается ограниченное определение подтипов и производных типов, язык Ada не позволяет расширить существующие типы посредством добавления в них новых атрибутов и методов. Это приводит к необходимости дублирования кода и накладывает серьезные ограничения на какие бы то ни было преимущества, получаемые от расширения. Кроме того, этими последними преимуществами трудно воспользоваться из-за присущего этому языку раннего связывания. Большое внимание, которое уделяется в языке Ada раннему связыванию и строгой типизации, вызвано требованиями эффективности и безопасности.

Благодаря, главным образом, постановлениям Министерства Обороны Ada становится основным языком, применяемым в большинстве оборонных проектов США; однако попытка проникнуть в коммерческие сферы закончилась неудачей. Обычно язык Ada критикуют, в основном, за то, что он не пытается решить проблемы, связанные с отсутствием объектно-ориентированного подхода. Этот язык обладает очень большими возможностями, и его критики утверждают (имея на то некоторые основания), что полный набор примитивных команд очень усложняет изучение и надлежащее использование этого языка. Кроме того, язык Ada плохо выглядит на фоне "экономных" языков, таких как C++, если взглянуть на него с точки зрения требований к памяти, которая необходима для работы компилятора.

Самая последняя версия языка (Ada 95), помимо его стандартных свойств (табл. 3.7), позволяет использовать объектно-ориентированные конструкции, например иерархии классов, простое наследование, динамическое связывание и перекрытие методов как на уровне классов, так и на уровне экземпляров. В языке Ada 95 (лучше, чем в Ada 83) поддерживается также создание прототипов и предусмотрены автоматические средства трассировки.

Язык Object Pascal (Apple) можно считать преемником языка Clascal; он использовался при разработке машины Apple Lisa и интерфейса Macintosh. По сути, этот язык представляет собой просто расширенный набор средств языка Pascal. Объекты в языке Object Pascal — это записи языка Pascal, в которых инкапсулированы процедуры и функции. Язык Object Pascal поддерживает только одиночное наследование и не имеет встроенной функции сборки мусора. Язык Delphi компании Inprise дополняет объектно-ориентированный язык Pascal, являющийся расширением языка Pascal, многими объектно-ориентированными свойствами, но абсолютно не поддерживает множественное наследование. В конечном счете расширения языка Pascal будут приветствоваться только теми организациями, которые уже вложили большие инвестиции в код, написанный на языке Pascal, и хотят перейти к объектно-ориентированному программированию. В настоящее время существуют версии языка Delphi, использующие, вместо Pascal, языки Java или C++.

Таблица 3.7. Свойства языка Ada

Поверка типов/связывание	Раннее (позднее — для типов записей в Ada 95)
Полиморфизм	Да
Сокрытие информации	Да
Параллелизм	Сложно
Наследование	Да, начиная с версии Ada 95
Множественное наследование	Нет
Сборка мусора	Нет
Перманентность	Нет
Универсальность	Да
Библиотеки объектов	Немного

Язык CLU, разработанный в конце 70-х годов компанией MIT [490], в свое время являлся передовым языком программирования. В этом языке основной единицей абстракции является *кластер* — понятие, аналогичное понятию класса в языке Simula. Но, в отличие от языка

Simula (по свой природе похожего на Algol), в CLU процедуры и структуры данных не могут задаваться вне кластера. Язык CLU снабжен отдельной формальной спецификацией языка, спроектированной специально для него и базирующейся на классах. Метод проектирования вызывает операции для описания алгебры каждого типа данных и поддерживает универсальность. Возможно наследование между классами и их объектами, но не наследование между метаклассами и их подклассами.

Еще один объектно-ориентированный язык программирования — Lingo, разработанный в Стратклидском Университете (Strathclyde University), сформировался на базе машины Linn Rekursiv [360, 639], которая вместе с машиной NeXT представляла, вероятно, первое поколение объектно-ориентированных вычислительных машин. Компании IBM и Apple учредили компанию Taligent, которая предназначалась для создания объектно-ориентированной машины и операционной системы с параметрами машины NeXT, надеясь при этом достичь большей совместимости с другими системами и стандартами. Проект явно провалился ввиду некоторых коммерческих и технических причин. Проект CommonPoint компании Taligent предусматривал свыше 100 каркасов, состоявших примерно из 4000 классов и 53000 методов. Это можно было бы сравнить, скажем, с программным интерфейсом приложений (API) операционной системы Windows 3.0, имеющим всего лишь 1500 вызываемых функций. Такая сложная архитектура, в сочетании со слабыми сторонами объектной модели языка C++, была широко раскритикована за провал данного рискованного предприятия [745].

Кроме того, можно было бы предъявить претензии и к операционным системам машин IBM System 38 и AS/400, как принадлежащим к нулевому поколению, поскольку на их проектирование несомненно повлияли объектно-ориентированные идеи. И опять же, самым правильным эпитетом для подобных разработок является, пожалуй, "объектный".

3.3. Функциональные и аппликативные языки программирования

Некоторые авторы подчеркивали разницу между объектно-ориентированным программированием и программированием, ориентированным на значения. В [504] обращается внимание на то, что значения (например, число 17) являются аппликативными и предназначены только для чтения; они представляют собой абстракции, никак не связанные со временем. Объекты (т.е. экземпляры) существуют во времени и могут создаваться, удаляться, копироваться, совместно использоваться и обновляться. Ссылки на значения прозрачны, т.е. при любой ссылке на них всегда будет получено одно и то же значение. В частности, на этом основаны главные критические замечания о подходе, принятом в языке Smalltalk, где абсолютно все рассматривается как объект. Неспособность осуществить правильное разграничение приводит к несколько рискованным последствиям. Структуры данных, по случайности оказавшиеся совместно используемыми, могут быть ошибочно скорректированы; или же существует опасность дорогостоящих непроизводительных издержек, связанных с дублированием. Почему при таких преимуществах аппликативного или ориентированного на значения программирования мы все же предпочитаем объекты?

Надо сказать, моделирование окружающей реальности в компьютерной системе является чрезвычайно упрощенным, если используемые структуры данных соответствуют сущностям реального мира. Файлы являются объектами и не могут быть описаны с помощью алгебраических выражений. В обычных языках программирования переменные являются объектами,

которые идентифицируются и различаются в соответствии со своими адресами в памяти не зависимо от содержащихся в них текущих значений. Считается, что значение принадлежит некоторому типу, подобно тому как объект принадлежит некоторому классу. Существенное отличие состоит в том, что два объекта, имеющие одинаковое описание, могут быть различными, в то время как не может быть двух значений с одним и тем же описанием. Например может существовать только одно целое число 17, но мы можем располагать двумя идентичными копиями Mona Lisa. То, что обратное высказывание неверно, может быть подтверждено рассмотрением двух объектов, имеющих разные описания, — утренней и вечерней звезд которые представлены одним физическим объектом — планетой Венера. Многие языки по добная проблема заводит в тупик. Например, файлам в языке Pascal не могут быть присвоены значения, и они не могут быть использованы в выражениях, даже если и объявляются точно так же, как и другие типы. Программисты, моделирующие системы, должны рассматривать состояние и изменение состояния, они должны учитывать время и (по крайней мере, в области искусственного интеллекта) возможность других сфер применения. Аппликативное программирование не предназначено для этой цели — оно имеет дело с математическими абстракциями, никак не связанными со временем.

Таким образом, необходимо четкое разделение объектов и значений, а также существование языка, поддерживающего такое разделение и допускающего использование соответствующих моделей (значений или объектов).

Примерами стиля функционального программирования могут служить языки, основанные на формальной логике и математике, например языки Lisp или ML. Во время работы обычных программ значения присваиваются переменным, которые представляют ячейки памяти. Любое значение, находившееся в этих же ячейках до сохранения данного значения, будет перезаписано и потеряно навсегда, если предварительно не предпринять определенных действий. Аппликативные языки, например язык Lisp, отказались от такого деструктивного процесса присваивания. В отличие от императивного программирования, в аппликативном программировании запрещены присваивания или побочные эффекты. На практике это означает, что процессор должен периодически выполнять некоторую сборку мусора, освобождаясь от значений, которым больше не следует находиться в памяти. Еще одним общим свойством являются *отложенные вычисления*[2], согласно которому значения не вычисляются до тех пор, пока функция их не затребует. Такие языки базируются на функциональных приложениях композиции и зависят от логической системы, известной как "лямбда-исчисление"; она будет рассмотрена ниже. Аппликативное программирование становится функциональным, когда оно поддерживает "прозрачность ссылок". Это означает, что каждое выражение или переменная имеет одно и то же значение внутри данной области видимости, т.е. все переменные являются локальными. Таким образом, всегда можно заменить одно выражение другим имеющим такое же значение, не изменив при этом значение всего выражения. Это свойство хорошо использовать для доказательства теорем и выполнения запросов к базе данных, где перезапись выражений и подстановка являются основными операциями. Хотя это правило заставляет вспомнить объектно-ориентированные языки, функциональное программирование не поддерживает состояние объектов.

Процедурный язык сообщает компилятору, каким образом нужно выполнить определенное задание. Непроцедурный язык программирования сообщает компилятору только *что* он должен делать, а не *как* это сделать. Рассмотрим следующий запрос к базе данных:

---

[2] *Их иногда называют "ленивыми вычислениями".* — Примеч. пер.

предполагающий получение количества сотрудников в каждом подразделении, а также их общей и средней зарплаты (в рамках подразделений).

```
SELECT    DNAME,  JOB,  SUM(SAL),  COUNT(*),  AVG(SAL)
FROM       EMPLOYEE, DEPARTMENT
WHERE    EMPLOYEE.DEPTNO = DEPARTMENT.DEPTNO
GROUP  BY     DNAME,  JOB
```

На самом деле этот запрос позволяет получить правильный результат, однако это нас сейчас не интересует. В данном случае речь идет о языке SQL. Заметим, что компьютер нигде не сообщает, *как* ответить на этот вопрос. Ответ предполагал бы получение списка сотрудников, отсортированного по подразделениям и выполняемой работе, и последующее вычисление их количества, а также средней и общей зарплаты. И наконец, в выходных данных название подразделения должно быть заменено на его номер. Это была бы достаточно сложная процедура, включающая чтение записей и сохранение промежуточных результатов на каждой стадии выполнения. В непроцедурном языке SQL, основанном на реляционном исчислении (мы рассмотрим его в главе 5), все это становится ненужным.

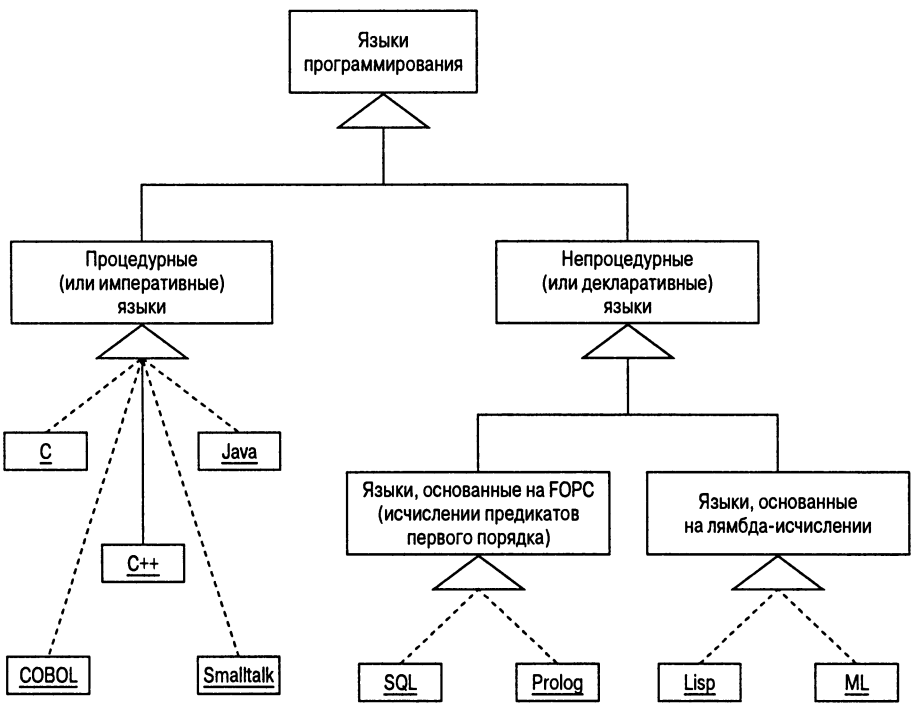


Рис. 3.1. Классификация языков программирования

Современные версии языка SQL содержат встроенные функции, благодаря которым вводится некоторая доля процедурности. В чистом виде это преимущество встречается редко.

Термин *декларативный* является немного более общим, чем *непроцедурный*, поскольку он предполагает использование чисто описательных языков. Основное внимание уделяется способу представления данных, а не какому бы то ни было конкретному стилю программирования. Императивный язык часто рассматривается как противоположность декларативному. Но его принятие часто тормозится ввиду стремительного наступления более модного, но менее благозвучного "непроцедурного". Язык Prolog также был выдержан в описательном и непроцедурном стиле (хотя содержал операцию вырезания (!), которая с таким же успехом могла быть использована и в "топорном" процедурном стиле). Давайте теперь обратимся к логике, лежащей в основе подобных языков.

Оба языка — и Prolog, и SQL — основаны на исчислении предикатов первого порядка, т.е. на некотором виде математической логики; при этом язык SQL базируется на меньшем подмножестве и поэтому менее выразителен. Такие языки обычно называются логическими языками программирования только в том случае, если они так же выразительны, как язык Prolog. Функциональные языки базируются на другой логической системе — лямбда-исчислении, и обычно их не называют логическими языками программирования, хотя они строго придерживаются логики.

Существует ряд особых преимуществ, которые традиционно связываются с функциональными языками программирования.

- формальная основа;
- они легко расширяются посредством добавления новых многократно используемых функций;
- единообразная идеология программирования (абсолютно все считается функцией);
- достаточно просто описываются конструкции более высокого порядка (функции функций);
- поддержка полиморфных типов (как в языке ML) или полностью лишенный типов стиль (как в языке Lisp).

Классификация языков программирования представлена на рис. 3.1.
Остальная часть этого раздела посвящена более углубленному (чем в оставшейся части главы) рассмотрению функционального программирования, поэтому специалисты, испытывающие отвращение к математическим терминам, могут ее опустить. Последующая часть книги будет понятна и без приведенных ниже подробностей.

Лямбда-исчисление возникло в 30-х годах исключительно благодаря Алонсо Черчу (Alonzo Church). Его первоначальной целью являлось предоставление базы для всей математики, основанной на правилах. При этом функция рассматривается как процесс перехода от аргумента к значению, а не в более современном смысле Дирихле (Dirichle), рассматривающего функции как графы (подмножества отношений). Можно считать, что такие функции или правила задаются собственными предложениями языка, используемыми с аргументами (также выраженными словами языка), или что эти функции или правила являются компьютерными программами, которые могут использоваться другими компьютерными программами или структурами. В обоих случаях язык не имеет типов. Говорят, что в нем нет различия между разными видами объектов: теми, которые образуют функцию, и теми, которые могут быть ее потенциальными аргументами. В частности, функция может использовать

саму себя, т.е. поддерживается рекурсия. В обычной теории множеств это не допускается, и поэтому описанные выше условия моделируются другой логикой исчисления предикатов первого порядка FOPC (First Order Predicate Calculus). Первоначальная теория Черча (Church) оказалась несовместимой с открытым позднее парадоксом Клина-Россера (Kleene-Rosser), который лежал в основе использования теории, на которой базируются современные исследования и некоторые из языков программирования. Самым известным среди них был язык Lisp, впервые появившийся примерно в 1956 году [533].

Поскольку язык Lisp (обозначающий List Processing — обработка списков) базируется на лямбда-исчислении, в нем нет различия между программой и данными: все они представлены в виде списков, имеющих иерархическую структуру.

Лямбда-исчисление представляет класс частично вычисляемых функций, т.е. функций, определенных не для всей своей области, а для целых чисел, которые оказываются рекурсивными функциями, и они эквивалентны вычисляемым функциям Тьюринга. На основе этой эквивалентности гипотеза Черча-Тьюринга утверждает, что обычное интуитивное понятие функции, которая может быть вычислена с помощью конечного алгоритма, эквивалентно понятию рекурсивной функции. Поэтому не вызывает удивления тот интерес, который проявляют к этой теме ученые, специализирующиеся в области компьютерных технологий. Вскоре Черч (Church) открыл, что лямбда-исчисление является неразрешимым. Проблема заключалась в существовании нормальной формы термов лямбда-исчисления. На языке машин Тьюринга они являются функциями, которые не имеют неприятных бесконечных циклов и эквивалентны "удовлетворительным" машинам Тьюринга.

В целях обеспечения полноты Черч ввел понятие оператора абстракции "лямбда", который позволяет создавать или *абстрагировать* функцию из элементов языка. Таким образом, (лямбда $x.M$)a представляет собой функцию, которая эквивалентна результату подстановки a для всех вхождений x и применения функции M. Например, можно создать функцию с одним аргументом, которая будет иметь следующий вид.

$$g = (\text{lambda } x.2*x+3).$$

Так что $g(1)=5$, $g(2)=7$, $g(3)=9$ и т.д. Кроме того, лямбда-исчисление поддерживает операцию, которая принимает значения "истина" (true) и "ложь" (false) и позволяет включать в определения функций условные выражения.

$$g = (\text{lambda } x.\text{if } x>5 \text{ then } f(x) \text{ else } -f(x)).$$

Абстракция функции и ее применение представляют собой базовые операции данной логики. Оператор "лямбда" универсальным образом расширяет тип языка, аналогично тому как инвертирование элементов x^{-1} расширяет язык теории элементарных групп или отрицательные числа расширяют арифметику натуральных чисел. Однако Карри (Curry) показал, что лямбда-оператор не является абсолютно необходимым, хотя интуитивно он "просится" в математический контекст. В современных диалектах языка Lisp явное обозначение этого оператора все чаще опускается, но он все еще присутствует неявным образом, о чем свидетельствует та осторожность, которую следует соблюдать при связывании переменных в сложных программах.

Формально язык лямбда-исчисления определяется так.

- Алфавит содержит переменные $x, y, z...$, а также символы редукции $\rightarrow$, равенства $=$ и абстракции λ (лямбда). Термы задаются рекурсивно следующим образом:

- переменные являются термами;
- применение двух термов дает в результате терм;
- если M является термом, а x — переменной, тогда (лямбда xM) представляет с бой терм.

Правильно построенные формулы задаются следующим образом: если M и N являют термами, тогда $M \rightarrow N$ и $M = N$ являются формулами. Операторы редукции, равенства и а стракции определяются при условии, что они удовлетворяют определенным системам аксио

Переменная является **несвязанной**, если она не находится в области видимости лямбда. в противном случае она является **связанной**. Таким образом, оператор лямбда эквивалент квантору всеобщности в логике предикатов или определенному интегралу системы исчисления

Все многочисленные диалекты языка Lisp основаны на лямбда-исчислении. Lisp в нек тором смысле является языком очень низкого уровня, базирующимся на весьма немног примитивах, которые можно рассматривать как виртуальную машину. К наиболее фундамен тальным примитивам можно отнести CAR, CDR (произносится как ко-дер), ATOM и LAMBD (или PROG). Единственным простым типом данных является список. Список содержит голов и хвост (физически — это указатель на голову другого списка). CAR возвращает голову к атом, а CDR — хвост как список; в современной терминологии они обычно называют "головой" и "остатком". Таким образом, головой списка `(яблоки, бананы, морковки` является элемент `(яблоки)`, а остатком — `(бананы, морковки)`. Атомы ссылаются н значения, находящиеся в памяти, и являются исходным материалом для построения списко Оператор CONS создает новый список посредством добавления в список новой головы. Кром того, здесь имеются некоторые управляющие структуры, а также арифметические и реляц онные операторы. Подобно языкам C и PASCAL, язык Lisp является рекурсивным.

Логическое программирование, примером которого могут служить различные диалект языка Prolog, основано на математической логике особого вида, известной как исчислени предикатов первого порядка (FOPC). Эпитет "первого порядка" имеет немаловажное значе ние, поскольку он отражает идею, согласно которой логика не может иметь дело непосредс венно с утверждениями об утверждениях, а только с утверждениями об объектах прими тивного (атомарного) типа. Хотя с помощью математики можно продемонстрировать, чт системы первого порядка могут выражать понятия более высоких порядков, но такое выра жение часто бывает бесполезным и весьма неестественным. Функциональное программи рование тоже может считаться некоторой формой логического программирования, однако его основе лежит другая логика — лямбда-исчисление, описанное выше. Хочется высказат одно интересное замечание, касающееся обоих видов логического программирования: они ба зируются на формальной теории, тогда как объектно-ориентированное программировани основано исключительно на модельном представлении и абсолютно лишено формальног обоснования. Это значит, что очень сложно доказать что-то относительно объектно ориентированных приложений. Не прекращаются попытки исправить этот явный недостато Язык EQLog [299] и его дочерний язык OBJ2 [286] базируются уже на логике другого вида – эквациональной логике; язык OBJ2 сознательно пытается найти единую формальную логик для логического и объектно-ориентированного программирования [299]. Еще одной тако попыткой можно считать язык FUN [141]. Мы рассмотрим некоторые результаты этих иссле дований в разделе 3.6.2. Однако мне кажется, что недостаток формальной теории не тольк не является препятствием для коммерческих разработок, но и может быть на практике пози тивным преимуществом, что мы и покажем в данной книге, особенно в главе, посвященно

базам данных. К другим важным функциональным языкам можно отнести язык ML, созданный Милнером [561], язык Hope [132], а также язык Miranda, автором которого является Тернер [762].

В научной литературе приводилось описание нескольких попыток объединения объектно-ориентированного программирования с функциональным и логическим. Основная трудность такого объединения состоит в том, что логическое программирование, основанное на FOPC (исчислении предикатов первого порядка), оказалось в принципе несовместимым с природой объектов, хотя абстракция и наследование могут быть смоделированы достаточно легко.

Однако одним из самых старых и популярных функциональных языков программирования является Lisp. В следующем разделе будут рассмотрены различные объектно-ориентированные расширения языка Lisp.

3.4. Системы, основанные на идеях искусственного интеллекта

В данном разделе рассмотрение объектно-ориентированных разработок, основанных на идеях искусственного интеллекта, подразделяется на две части. Вначале рассматриваются расширения языка Lisp, а затем — платформы разработки современных экспертных систем, причем не все из них базируются на языке Lisp.

3.4.1. Расширения языка Lisp

На протяжении многих лет Lisp являлся основным языком, применяемым во время академических исследований в области искусственного интеллекта. Язык Lisp предоставляет многое из того, что необходимо для реализации объектно-ориентированного языка в стиле Smalltalk. В нем предусмотрены сборка мусора, динамическое связывание, редакторы, отладчики и однородный лишенный типов стиль. Кроме того, разработаны многочисленные повторно используемые функции, и некоторые из них являются общедоступными. Поэтому вполне естественным выглядит наличие нескольких объектно-ориентированных расширений языка Lisp.

Влияние искусственного интеллекта ощущается, главным образом, в системах наследования некоторых расширений языка Lisp, где могут быть унаследованы не только атрибуты, но и их значения; как правило, поддерживается множественное наследование и демоны, т.е. процессы, управляемые событиями. Демоны (или триггеры, как они называются в литературе по базам данных) представляют собой операции, которые связываются со структурами данных и запускаются при обращении к этой структуре. Таким образом, существуют демоны, выполняемые при "необходимости" и при "обновлении". Эти два типа демонов иногда называются демонами прямого и обратного вывода соответственно. Такие операции обычно связываются не с объектами, а с атрибутами объектов, что можно рассматривать как выделение этих атрибутов в виде отдельных объектов с собственными правами или как нарушение принципов инкапсуляции — в зависимости от того, являетесь ли вы противником или сторонником систем такого рода. Главное, что можно сказать насчет такого подхода (как будет показано далее), — это то, что он в большей степени охватывает семантику приложения.

В настоящее время стандартом для расширений языка Lisp является CLOS — система объектно-ориентированного программирования на языке Common Lisp [568]. CLOS представляет собой расширение языка Lisp, основанное на понятиях родовых функций, множественного

наследования и объединения методов. Метаобъекты позволяют пользователю изменять базовую структуру самой системы объектов [444]. Эта система была спроектирована по подобию языка Smalltalk в PARC (Исследовательский центр компании Xerox в Пало-Альто).

В системах искусственного интеллекта понятием, в наибольшей степени соответствующим понятию объекта, является фрейм. **Фрейм** (frame) представляет собой структурную абстракцию, которая воплощает идею стереотипного объекта. Фреймы содержат расширяемые списки **слотов** (slot), которые эквивалентны атрибутам (или переменным класса). Слоты могут содержать описание как состояния, так и процесса. Со слотами можно связывать методы. Слоты, содержащие состояния, могут иметь также **фацеты** (facet) (в дополнение к значениям), которые являются методами для определения поиска, значений по умолчанию и триггеров. В CLOS каждый объект обладает уникальным идентификатором и слоты содержат указатели на другие объекты, представляющие состояние всего объекта в целом. Каждый экземпляр принадлежит некоторому классу, и каждый класс представляет собой тип. Слоты могут иметь определенный тип либо могут быть свободными; они могут принимать либо значения определенного типа, либо любое значение — независимо от какого то бы то ни было типа. В языке Common Lisp родовые функции действуют как объекты нескольких типов. Методы в CLOS позволяют определить, что должны делать родовые функции при вызове с определенными аргументами. Соответствие между родовыми функциями и относящимися к ним наборами методов является частью системы соответствия между функциональным и объектно-ориентированным программированием. Методы определяются посредством задания родовой функции, условий, при которых она применяется к объектам, параметров, правил наследования и кода.

Существуют методы нескольких типов. Методы **доступа** осуществляют доступ к (или обновляют) фацету значения слота. Это специальные методы (они упоминались в главе 1), которые позволяют говорить непосредственно об атрибутах в объектно-ориентированной среде. Это означает, что реализация состояния скрыта и обеспечивается с помощью стандартных средств доступа — чтения и записи значения. Эта концепция будет играть главенствующую роль в главе 6.

Между CLOS и многими объектно-ориентированными языками программирования существует одно важное различие. Этот язык позволяет определять для фацетов (т.е. подслотов) не "значения", а другие данные. Атрибут или слот некоторого экземпляра может иметь значение. Например, значением слота `BirthDate` (день рождения) для сотрудника `Employee` "Ian Graham" может быть величина `19480813`. Правильно созданный класс может иметь унаследованные атрибуты, имеющие значения по умолчанию; так что если подразделение, в котором работает указанный на рис. 3.2 сотрудник, неизвестно, можно предположить, что этим подразделением будет отдел продаж. В CLOS в подклассы можно добавлять свои слоты, методы и типы, а также условия инициализации (дополняя унаследованные ими от суперклассов). В этом отношении система CLOS похожа на язык Smalltalk. Ввиду тесной связи между языком Lisp и искусственным интеллектом не вызывает удивления тот факт, что CLOS поддерживает множественное наследование. Подобно языку Flavors — более раннему объектно-ориентированному расширению языка Lisp — конфликты наследования в CLOS разрешаются посредством задания порядка старшинства в списке суперклассов, указываемых при определении класса. Это достаточно гибкая система, но не настолько общая, как описанная в главе 6 схема, предназначенная для общей объектно-ориентированной спецификации. Список старшинства классов определяет наследуемые свойства слота в случае возникновения конфликта имен. При наследовании более чем одного метода в CLOS можно объединять методы при помощи сложной технологии, именуемой "комбинацией методов", которая основана на

существующем в языке Flavors понятии "смеси". Как сделать это наилучшим образом — еще не полностью разрешенная в настоящее время проблема, хотя значительные успехи продемонстрировал подход, принятый в других объектно-ориентированных языках. Мы возвратимся к этому вопросу в главе 6, где не будут рассматриваться конкретные языки. Слоты не только могут иметь значения по умолчанию, но и могут быть методами по умолчанию, которые могут быть перекрыты. Демоны прямого и обратного вывода могут задаваться для имитации пред- и постусловий, триггеров и побочных эффектов. Подобно языку Smalltalk, независимо от рассмотрения всего содержимого в качестве объектов, CLOS предусматривает собственные языковые конструкции, включая структуры под названием *метаобъекты*, обеспечивающие чрезвычайно хорошие возможности для расширения непосредственно самого языка. Подобно всем остальным системам Lisp, CLOS производит сборку мусора, но иногда в неподходящие моменты. Динамическое связывание поддерживается с помощью того же механизма, который используется в языке Lisp для выделения памяти на стадии выполнения. Так как имена слотов инкапсулированы, но не скрыты, классы CLOS не являются истинно абстрактными типами данных, однако этот недостаток можно преодолеть посредством хорошего стиля программирования.

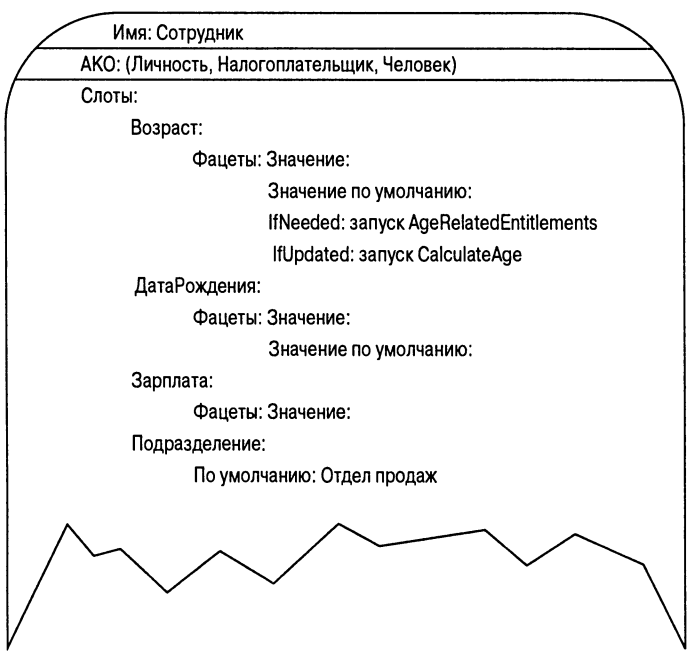


Рис. 3.2. Часть фрейма (не обязательно фрейма CLOS) для объекта `Employee` (служащий). Заметим, что каждый слот имеет фацеты для своего значения, значения по умолчанию и демоны, которые вызываются при обновлении и при необходимости. Демоны обновления представляют собой процедуры (или правила), которые запускаются всякий раз, когда фацет значения (или значения по умолчанию) изменяется. Демон, запускаемый при необходимости, ищет значение для заполнения фацета значения, если он доступен и не содержит значения в данный момент

С помощью CLOS разработаны разнообразные приложения: системы анимационной графики для коммерческого телевизионного вещания, географические информационные системы и системы управления телефонной сетью (табл. 3.8).

Таблица 3.8. Свойства CLOS

Контроль типов/связывание	Позднее
Полиморфизм	Да
Сокрытие информации	Да
Параллелизм	Затруднен
Наследование	Да
Множественное наследование	Да
Сборка мусора	Да
Перманентность	Нет
Универсальность	Да
Библиотеки объектов	Немного

Различные объектно-ориентированные расширения языка Lisp постепенно появлялись на протяжении 1970-х и 1980-х годов в среде специалистов по искусственному интеллекту. Язык LOOPS [88], язык Common LOOPS, разработанный компанией Xerox [90], а также языки Flavors, KEE и NewFlavors — все это типичные попытки такого рода. Система CLOS представляет собой стандарт, который объединяет все эти подходы в рамках языка Lisp.

Среди подобных расширений особенно интересным представляется язык CommonObjects, разработанный Снайдером [720]. В данной системе большое значение придается реализации наследования без нарушения инкапсуляции. При этом не разрешается использовать наследование для атрибутов — могут наследоваться только методы. Альтернативным и, как я полагаю, более хорошим вариантом является существование двух отдельных иерархий наследования: для методов и атрибутов (см. работу [210]) или, что еще лучше, для интерфейса и реализации, как в языке Java. Понятие наследования в области искусственного интеллекта и культура языка Lisp противоречат подобной защите, и, как мы уже выяснили, протоколы защиты, предложенные Снайдером, могут быть реализованы скорее в рамках хорошего стиля программирования, а не средствами компилятора. Эта проблема относится к разряду нерешенных и наводит меня на мысль, что объектно-ориентированное программирование является все еще в некоторой степени незрелой дисциплиной.

Число различных объектно-ориентированных расширений языка Lisp слишком велико, поэтому все они не могут быть даже перечислены здесь, не говоря уже об описании. В дополнение к уже упомянутым можно указать язык Ceyx, разработанный компанией INRIA, язык Oaklisp, авторами которого являются Ланг и Пельмуттер [467], а также многие другие.

Некоторые среды программирования более высокого уровня, предназначенные для выполнения подобных задач, появились в качестве средства разработки экспертных систем на основе искусственного интеллекта. Однако не все они базировались на языке Lisp. К рассмотрению именно таких средств мы сейчас и приступим.

3.4.2. Другие системы разработки, основанные на идеях искусственного интеллекта

KEE и ART представляют собой среды разработки экспертных систем на базе языка Lisp, первоначально спроектированные для специализированных рабочих станций. Они в определенной степени выдержаны в объектно-ориентированном стиле. При рассмотрении этих двух языков в качестве модели будет выступать язык KEE. Он претендует на звание объектно-ориентированного, главным образом, благодаря тому, что его **элементы** поддерживают наследование и множественное наследование атрибутов и методов. Слоты в этих элементах содержат *фацеты*, которые могут быть использованы для определения режима управления, при котором происходит наследование, например, чтобы определить, будут ли унаследованные значения иметь приоритет над значениями по умолчанию или будет ли допускаться множественное наследование. Однако язык KEE не так строг по части инкапсуляции, и пользователь может получить из языка Lisp прямой доступ к состоянию элемента.

Языки Kappa и Nexpert Object пытаются предоставить аналогичные возможности в среде, отличной от среды языка Lisp. Второй язык поставляется в форме библиотеки классов. Языки Kappa и Nexpert Object написаны скорее в стиле языка C, а не Lisp, и последствия этого сказываются, главным образом, на производительности. Рассматривая язык Nexpert в качестве представителя такого типа, отметим, что хотя для классов и поддерживается множественное наследование, объект может быть экземпляром не более чем одного класса. В целом эти пакеты обнаруживают свойства, сходные с языками KEE и ART. Обе системы — ES/KERNEL компании Hitachi и XShell компании Expersoft — представляют собой оболочки экспертных систем, имеющие в значительной мере объектно-ориентированное содержимое. Вторая система написана на языке C++, и в ней особое внимание уделяется распределенной реализации. Необычность этих двух продуктов заключается в том, что в них предусмотрены средства включения нечеткой логики. Язык ObjectIQ представляет собой латинизированную версию системы ES/KERNEL, у которой эти средства нечеткой логики отсутствуют.

Язык Aion, разработанный компанией CA, представляет собой среду разработки экспертных систем на основе продукционных правил. Эта среда поддерживает также объектно-ориентированное программирование. В компании CA для языка Aion была разработана технология моделирования, основанная на методе проектирования Catalysis и принципах инженерии знаний KADS. Концепция этой технологии весьма схожа с идеями объектов-агентов, основанных на системе правил, которые будут представлены в главе 6; однако здесь нет отдельного понятия правила, инкапсулированного в объекте, и методы подразделяются скорее на методы логического вывода и процедурные методы. Тем не менее, как и почти во всех оболочках экспертных систем, здесь внимание фокусируется на правилах, а не на объектах.

Все системы, появившиеся в результате разработок в области искусственного интеллекта, поддерживают наследование, иногда в ущерб инкапсуляции. Кроме того, они предлагают обладающую широкими семантическими возможностями форму наследования, которая отличается от той формы, которую можно обнаружить почти во всех объектно-ориентированных языках программирования, подобных языку Smalltalk. Хотя в языке C++ разрешено наследование значений по умолчанию, в некоторых объектно-ориентированных языках дела обстоят иначе. В системах, принадлежащих области искусственного интеллекта и основанных на фреймах, значения (в частности, значения по умолчанию) могут наследоваться подобно именам атрибутов. Это значит, что использование множественного наследования является более сложной проблемой и лишь отражает общий принцип, гласящий, что семантически сложные

системы предъявляют более жесткие требования к уровню квалификации программистов и разработчиков. Я полагаю, что специалисты в области искусственного интеллекта движутся в правильном направлении и что наследование такого рода в конечном счете должно войти в мир объектно-ориентированного программирования. Специалисты по объектно-ориентированному программированию многое должны почерпнуть из других компьютерных дисциплин, включая искусственный интеллект, и, как было отмечено Роджером Кингом [450], из работ в области семантического моделирования данных. В этой книге мы постоянно будем возвращаться к этой теме.

В настоящее время не существует идеальных объектно-ориентированных средств разработки экспертных систем, хотя некоторые очень хороши. Одной из причин можно назвать потребность в реализации эффективного "разумного" алгоритма, похожего на алгоритм RETE, который легче реализовать тогда, когда объекты создаются в виде побочного продукта правил обработки, а не тогда, когда правила связываются с объектами в экспертных мини-системах.

Таким образом, мы увидели, что среди огромного количества языков, претендующих, в некотором роде, на звание объектно-ориентированных, существуют значительные различия в смысле выделения определенных свойств и функциональности. Возникает подозрение, что использование термина "объектно-ориентированный" часто вызвано маркетинговыми соображениями. Иными словами, стало модно почти любой продукт описывать как объектно-ориентированный или основанный на компонентном подходе, чтобы создать иллюзорное разграничение между собой и конкурентами. Журналисты называют такую традицию "пускать пыль в глаза".[3]

3.5. Объектные библиотеки, каркасы приложений и объектно-ориентированные языки программирования четвертого поколения

Объектная библиотека представляет собой коллекцию завершенных, протестированных, документированных, многократно используемых объектов, распространяемых либо на коммерческой основе, либо как часть библиотеки программного обеспечения, разработанной собственными силами. Для объектно-ориентированных разработок объектные библиотеки имеют ключевое значение, поскольку без них доступны очень немногие преимущества многократного использования. Как правило, объектно-ориентированные языки программирования поставляются с базовыми библиотеками родовых объектов, но обычно их недостаточно для эффективной разработки приложений. Поэтому сейчас существует множество доступных коммерческих библиотек, которые могут служить дополнением языка; одним из популярных примеров является библиотека Rogue Wave.

Среда ObjectWorks компании ParcPlace Systems — это одна из первых систем, основанная на языке C++ или Smalltalk. Версия, предназначенная для языка C++, предоставляет интерфейс для операционной системы Unix, обеспечивает инкрементную компиляцию, отладку на уровне исходных текстов и графические средства просмотра/редактирования. Среда обеспечивает безопасность типов и поддерживает множественное наследование, что не всегда предоставляют языки C++ и Smalltalk. Версия, предназначенная для языка Smalltalk 80, включает библиотеку, содержащую свыше 300 классов и несколько тысяч методов.

[3] *Caveat emptor — пусть покупатель будет бдителен!*

Не нуждаясь в изменениях, она работает на самых разных машинах — от персональных компьютеров до рабочих станций, функционирующих под управлением Unix — и совместима на уровне исходного кода по всему этому диапазону.

В настоящее время получили широкое коммерческое распространение объектные библиотеки, предназначенные для создания графических пользовательских интерфейсов на языке C++. Существуют также библиотеки, совместимые почти со всеми платформами разработки, например MFC (библиотека базовых классов) в рамках среды Visual Studio компании Microsoft. Как правило, все эти библиотеки содержат классы, представляющие абстракции программирования довольно низкого уровня, например стеки или связанные списки, и графические понятия, например прямоугольники, кнопки или окна. Начинают появляться библиотеки, классы которых нужны для разработки специальных коммерческих систем; одним из первых примеров может служить библиотека San Francisco, разработанная компанией IBM.

Существует несколько объектно-ориентированных **каркасов приложений** (application framework) для самых распространенных типов компьютерных систем. Такие каркасы способствуют повышению производительности приложений, в которых активно используется графический пользовательский интерфейс, доступ к базам данных и взаимодействие на основе архитектуры "клиент/сервер". Помимо обычных базовых классов низкого уровня, они как правило, предлагают средства визуального программирования и обширные библиотеки классов для программирования окон и интерфейсов баз данных. Иногда они включают классы для генерации отчетов и другие полезные инструменты.

LispWorks представляет собой коммерческую среду программирования для рабочих станций в рамках CLOS (системы объектно-ориентированного программирования на языке Common Lisp). Доступ к X-окнам осуществляется через расширяемый набор виджет-классов, т.е. классов предварительно определенных элементов окон, соответствующих данному аппаратному и операционному обеспечению.

Компания Sun создала среду разработки приложений на базе репозитариев Forte, которая включает истинно объектно-ориентированный язык программирования четвертого поколения TOOL. Она также содержит интерфейсы для реляционных баз данных, поддерживает технологию CORBA и программное обеспечение среднего уровня, а также генератор графических пользовательских интерфейсов. PCTE-совместимый репозитарий создан на базе ObjectStore.

Без объектных библиотек программирование становится чрезвычайно трудоемким. Одной из сильных сторон языка C всегда была его способность поддерживать библиотеки функций. Добавление библиотек объектов является естественным продолжением развития языка, что, вероятно, повысит потенциал его коммерческого применения. Однако существует и негативная сторона использования библиотек того или иного вида, что на самом деле продемонстрировала история развития языка C. При каждом подключении частной библиотеки уменьшается переносимость приложения. Это зачастую не важно для конкретных приложений, но имеет большое значение при разработке пакетов. Более того, поскольку одной из главных причин использования объектно-ориентированного программирования является возможность многократного использования, приходится думать о том, не будет ли это преимущество уже само по себе скомпрометировано ввиду недостаточной переносимости. Разработчики систем должны рассматривать этот вопрос, требующий компромиссных решений, на ранней стадии, т.е. на этапе планирования каждого нового проекта.

Основная трудность, связанная с библиотеками классов, состоит в исключительной сложности понимания, *что* же содержат библиотеки и *что* делают находящиеся в ней классы. По оценкам специалистов, хорошему программисту нужен примерно один день на

то, чтобы полностью понять работу класса. Таким образом, чтобы ознакомиться с типичной библиотекой классов (в библиотеке языка Smalltalk содержится примерно 350 классов), потребуется более одного человеко-года. На фоне этого, опять же по оценкам специалистов, на создание полностью протестированного и отлаженного класса уйдет в среднем два человеко-месяца, т.е. для создания библиотеки, состоящей из 350 классов, потребуется примерно 60 лет. Конечно, нет такой программы, которая нуждалась бы сразу во всех классах библиотеки. Подобные наблюдения свидетельствуют о необходимости групповой работы. Командам следует регулярно собираться с целью повторного критического просмотра разрабатываемого кода. Преимущество такого подхода состоит в том, что не только разработчик, но и кто-то другой может обладать информацией о классе, который может быть использован для ускорения разработки. Методы групповой работы должны быть хорошо знакомы программистам, пишущим на языке FORTRAN и обычно работающим с большими библиотеками функций, например библиотекой NAG, содержащей математические процедуры языка FORTRAN. Эти методы важны и для объектно-ориентированного программирования.

При наличии больших объектных библиотек управление изменениями приобретает особенно важное значение. Проблема еще больше усложняется из-за существования сложных объектов, состоящих из нескольких взаимосвязанных объектов. Каким образом изменение одного компонента воздействует на всю систему в целом? Теоретически в идеально инкапсулированных языках изменение реализации не должно оказывать никакого влияния, однако если разрешено наследование, случай уже не является идеальным и, что еще хуже, мы никогда не сможем дать реальной гарантии, что интерфейс объекта не может быть изменен. Один из способов решения данной проблемы предложен в языках межмодульного соединения, например в DeRemer и MIL75 компании Kron. Они являются метаязыками, которые обеспечивают явную взаимосвязь между объектами. Для применения преимуществ многократного использования программного обеспечения, нужен язык компоновки, который мог бы применяться для задания интерфейсов модулей, чтобы многократно используемые модули можно было конфигурировать в соответствии со спецификацией и каждый модуль независимо от остальных имел бы свой язык реализации. Помимо этого нужны методы классификации объектов, чтобы помочь программистам разобраться с библиотеками, с которыми они работают. Таким образом, существует огромная потребность в инструменте, который мог бы оценить влияние изменений на интерфейс, т.е. прогнозировать изменение [394]. Полное отсутствие таких средств и методов в коммерческой сфере свидетельствует о незрелости объектно-ориентированного программирования. Однако появление таких средств может служить индикатором его развитости. В главе 2 мы уже обсудили тот факт, что управление компонентами представляет собой сложную и неразрешимую проблему.

Отсутствие библиотек классов высокого уровня для подавляющего большинства коммерческих приложений является серьезным препятствием на пути к повсеместному переходу к объектно-ориентированному программированию. Здесь наблюдается порочный круг: без таких библиотек разработки стоят слишком дорого, а хорошие библиотечные модули могут появиться только в результате выполнения реальных проектов. Компании, желающие получить преимущества от объектно-ориентированного программирования, должны обосновать стоимость проектов, предусматривая долгосрочное многократное использование, что особенно трудно достигается в период высоких процентных ставок. Это наводит на мысль о том, что первые коммерческие библиотеки будут продаваться компаниями, которые создали системы и намерены возместить некоторые свои затраты на разработку; и на начальном этапе эти библиотеки могут иметь вид пакетов прикладных программ, рассчитанных на использование в определенных условиях.

Покупатели таких пакетов потребуют еще большей гарантии того, что модули являются достаточно высококачественными, т.е. что они были спроектированы и разработаны с целью поддержки многократного использования уже на начальном этапе и продаются не только для возмещения затрат на разработку.

3.6. Другие направления развития

В этом разделе приводится обзор некоторых языков, которые оказались важными или повлияли на развитие обсуждаемых в данной книге идей. Здесь мы увидим, насколько осуществима реализация объектно-ориентированных идей при помощи обычных языков программирования. Большая часть тем, рассматриваемых в данном разделе, уводит в сторону от общих понятий объектно-ориентированного программирования, поэтому читатель может пропустить или бегло просмотреть этот раздел, что никак не повлияет на восприятие последующего материала книги.

3.6.1. Другие языки

Язык Trellis

Разработанный компанией DEC, язык Trellis/Owl [682] был весьма необычным, поскольку сочетал в себе свойства языка со строгой типизацией и системы управления данными. Он испытывал сильное влияние со стороны языка CLU и поддерживал множественное наследование, параллелизм и универсальность. Помимо этого, в нем использовалась модель данных. В главе 5 рассматриваются объектно-ориентированные и другие типы баз данных.

Объектная система Trellis была объединена с реляционной базой данных под управлением виртуальной машины на основе языка структурированных запросов SQL и включала библиотеку средств просмотра, отладки и программирования. В ней были предусмотрены процедура сборки мусора и перманентные объекты. Кроме того, осуществлялась поддержка параллелизма посредством управления множественными потоками, множественного наследования и перекрытия. Система Trellis поставлялась вместе с исходным кодом, однако компания DEC не поддерживала его в случае изменения. Позиция компании DEC по отношению к Trellis напоминала их же позицию по отношению к языку продукционных правил OPS-5, когда на протяжении многих лет они приобретали внутри своей компании определенные навыки и создали несколько крупных приложений на языке, который не получил широкого распространения на другом оборудовании. Его преимущества заключались в том, что язык был приспособлен к среде и существовала весьма основательная его поддержка. Недостатком, по крайней мере для пользователя, являлась потенциально возможная слабая переносимость.

Ряд других так называемых языков "перманентных объектов" описан в [667]. Язык PS-Algol — это версия языка Algol, использующая перманентные объекты; описана в [341].

Actor

Язык Actor [786] представляет собой объектно-ориентированный язык и среду, влияние модели активного объекта на которую сказалось, пожалуй, только на имени [6]. Эта среда была спроектирована для поддержки разработчиков, работающих под управлением операционной системы Microsoft Windows, и содержала обширную библиотеку классов, оснащенную

обозревателем и отладчиком. Этот язык поддерживает инкапсуляцию, простое наследование, позднее связывание, инкрементную компиляцию и сборку мусора. В 1992 году среда Actor была приобретена компанией Semantec.

Здесь не раз уже были упомянуты альтернативные объектно-ориентированные языки программирования и похожие идеи, которые очень тесно связаны с объектами, например системы на основе исполнителей. Точно так же, как в языке Smalltalk все элементы являются объектами, в Actor все элементы — исполнители. Они обладают уникальной постоянной индивидуальностью и текущим поведением, которое может меняться со временем и которое определяет, как исполнитель будет отвечать на следующее получаемое им сообщение. Поведение образуют переменные экземпляра, называемые *знакомыми*, и методы, именуемые их *сценариями*. Знакомые определяют других исполнителей, с которыми может связываться данный исполнитель. Исполнитель, не имеющий знакомых, является кандидатом на удаление сборщиком мусора. По получении сообщения текущее поведение может быть изменено. Если исполнитель не имеет метода для обработки данного сообщения, он может *делегировать* обработку *посреднику* (proxy) — наделенному этой обязанностью исполнителю, принадлежащему к кругу знакомых. Это понятие в чем-то похоже на понятие суперкласса, однако методы не наследуются, т.е. код не копируется; и единственное, что происходит — это передача сообщения. В отличие от объектно-ориентированного программирования, системы на основе исполнителей обладают строго определенной семантикой, но соответствующие языки являются языками очень низкого уровня и сложны в применении.

В дополнение к языкам на основе исполнителей (например, к языку Act 3) существуют также *параллельные* версии чисто объектно-ориентированных языков, например язык ConcurrentSmalltalk [813], и гибридные языки, например язык Orient84/K, сочетающий в себе стили языков Smalltalk и Prolog [409].

Такие языки, как Act 3 и ABCL/1, реализуют наследование без участия классов, поэтому трудно сказать, следует ли называть их объектно-ориентированными. На практике они предоставляют такие преимущества, которые характерны для инкапсуляции и наследования, и поэтому разделение выглядит нелепо, хотя такие языки демонстрируют особенности языков очень низкого уровня. Язык Actor является объектно-ориентированным, а не языком на основе исполнителей.

Языки BETA и Mjølner

Язык BETA сочетает в себе идеи ориентированного на ограничения логического программирования с объектно-ориентированным подходом. Он создавался в рамках исследовательского проекта, результаты которого сейчас доступны на коммерческой основе в рамках среды разработки Mjølner [454], распространяемой датской компанией Mjølner Informatic. Язык BETA [507] базируется на более общей модели, чем язык Smalltalk, и обеспечивает более широкие возможности концептуального моделирования, позволяя объектам и методам существовать отдельно от классов. Своим происхождением он обязан языку Simula и оказался значительным достижением в области объектно-ориентированного программирования, поскольку демонстрирует, скорее всего, перспективные направления в развитии языков.

В языке BETA объекты рассматриваются как физические объекты реального мира. Каждый из них обладает уникальной природой и может быть объединен с другими объектами. Делается четкое разграничение между частью чего-то и свойством чего-то, например "моя машина имеет колесо и цвет". В отличие от языка Smalltalk, числа являются значениями, а не объектами первого класса, таким образом решается вопрос, связанный

с основными критическими замечаниями по поводу языка Smalltalk. Самым мощным понятием языка является **шаблон.** Шаблон может представлять класс, структурированный тип или процедуру, например машину, запись в журнале или службу в радиусе 20000 миль. Шаблоны могут представлять практически любое явление, например переменные, структуры данных, подпрограммы или активизацию процедур. Объекты являются экземплярами, но не классов, а шаблонов, и именно шаблоны участвуют в схемах наследования. Интересно, что специалисты, работающие в области объектно-ориентированного анализа, совершенно независимо от других также открыли для себя пользу применения шаблонов. В [167] предлагается поискать повторяемость в структурах классификации и композиции, а в [417] наследование применяется не только для объектов, но и для прецедентов (см. главу 6).

Язык BETA был создан финской компанией Nokia Telecommunications, работающей в области цифровой телефонной связи. Существуют версии языков BETA и Mjølner, работающие в среде программных продуктов Microsoft, Macintosh и Unix. В состав системы Unix входит CASE-средство автоматизированного проектирования и создания программ для начальных этапов жизненного цикла разработки. Это средство содержит генератор кода и позволяет выполнять визуальное программирование; кроме того, программы могут быть преобразованы в диаграммы (обратного проектирования). В последующих главах будет показано, насколько важна возможность обратного проектирования при разработке программного обеспечения. Насколько известно автору, ни один другой язык не поддерживает подобную возможность. Как и в языке Smalltalk, здесь предусмотрены возможности метапрограммирования, при помощи которого может быть изменена сама среда.

Такие разработки указывают направление совершенствования самой объектно-ориентированной парадигмы, однако в настоящее время они не получили широкого коммерческого распространения, и язык BETA нечасто используется за пределами Скандинавии. Однако я полагаю, что хорошие идеи, заложенные в языке BETA, будут подхвачены другими языками и еще скорее — методами разработки.

Еще о некоторых языках

Язык Oberon был разработан Виртом (Wirth) как преемник языка Modula-2. Язык Modula-3 представлял собой аналогичную версию этого языка, разработанную компанией DEC, и имел похожие на классы структурные типы, основанные на модели языка Ada-95. Язык Sather [746] являлся аналитическим языком, который преследовал те же цели, что и язык Eiffel, но в большей степени фокусировал внимание на формализмах и подстановках. Язык Squeak [263] представлял собой попытку Алана Кэя (Alan Kay) снова вернуться к первоначальной версии языка Smalltalk, которая, как он считал, была искажена вмешательством коммерческих интересов. Это была принципиально новая реализация языка Smalltalk 80 на высшем уровне виртуальной машины, написанной на языке Smalltalk и связанной с объектно-ориентированной базой данных GemStone.

Предметы и аспекты

Появление предметно-ориентированного [363] и аспектно-ориентированного программирования [443] — SOP и AOP соответственно — вызвано слабостью объектно-ориентированного программирования, которая обнаруживается в том случае, когда необходимые свойства противоречат структуре классов и приводят к изменению интерфейсов. Мы знаем, что в объектно-ориентированных приложениях изменения локализованы в структурах данных

или реализации программы, однако изменения свойств имеют более серьезные последствия. Происхождение SOP (предметно-ориентированного программирования) связывают с идеей списков свойств языка Lisp. Состояние или поведение предмета связывают с его идентификатором. Затем различные предметы можно рассматривать как один и тот же объект, имеющий различные параметры поведения. Аспектно-ориентированное программирование (AOP) подразумевает, что разработчики должны отдельно кодировать различные "аспекты" программы, например безопасность, перманентность или вычисления. Тогда необходим инструмент, например, AspectJ [445] для объединения всех этих аспектов в единое целое. Компонентная разработка, рассматриваемая в главе 7, также имеет отношение к данной теме.

3.6.2. Теории типов и объектно-ориентированное программирование

Вплоть до настоящего времени объектно-ориентированное программирование развивалось в условиях отсутствия каких-либо строгих математических теорий в отношении синтаксиса или семантики. Объектно-ориентированное программирование — это скорее стиль или модельное представление, а не формальный метод, но это действительно очень мощный подход. Поэтому не раз делались попытки предоставить более строгое обоснование. Предпринимались попытки объединить объектно-ориентированное программирование с функциональным или логическим или обоими этими подходами к программированию вместе, а также попытки предоставить строгие теории типов (обзор таких подходов содержится в работе [210]) или даже семантику обозначений [185].

Общее свойство функционального, логического и объектно-ориентированного программирования состоит в том, что все они пытаются отделить описание вычислений от деталей его реализации. Они пытаются моделировать проблему реального мира, а не вычислительный процесс, что характерно для обычных языков, подобных языкам COBOL или FORTRAN. Функциональное и логическое программирование обычно основано на формальных принципах логики первого порядка, лишенной типов. Исследования в этой области привели к разработке алгебраических языков, например языка OBJ2 [286]. Эти усилия направлены также на создание доказуемо точных компиляторов. Теории, базирующиеся на алгебраических моделях, примером которых может служить язык OBJ2, предполагают строгое разграничение между значениями и операциями. В то же время теории, основанные на логиках более высокого порядка (примерами могут служить следующие языки: SOL [565], FUN [141], DL [505], Russel [224, 395] и Poly [530]), позволяют трактовать функции и даже непосредственно сами типы как аргументы (значения); иными словами, как объекты первого класса. В [1] сделана попытка уйти от обычных принципов, основанных на λ-исчислении, и ввести объектные исчисления, примитивными термами которых являются объекты.

Полезность теорий типов для языков программирования объясняется тем, что они позволяют компиляторам выполнить основную проверку на стадии компиляции и таким образом избежать многих ошибок на стадии выполнения. В [107] предлагается в языке Smalltalk в качестве типа объекта рассматривать его ближайший суперкласс. В языке Emerald [81] подтип принадлежит типу X при условии, что он предусматривает все операции типа X с тем же списком аргументов и возвращает результаты того же типа.

Подстановки связаны с проблемой возможности принятия решения, и этот вопрос становится еще более запутанным, когда вводятся множественные потоки и распределение. Многоуровневые архитектуры, допускающие обратный вызов, еще более усложняют задачу.

Интересные рассуждения на эту тему можно найти в работе [745]. Вся эта область — предмет глубоких и активных исследований, и потребуется некоторое время для того, чтобы все эти диспуты разрешились и привели к модели, согласующейся со всеобщими коммерческими устремлениями. А пока мы останемся с объектно-ориентированным модельным представлением, лишенным общепринятой формальной базы. Однако тот факт, что подобные формальные принципы уже привлекли внимание исследователей, вселяет надежду.

Я полагаю, что отсутствие формальной теории не является помехой для разработки коммерческих приложений, за исключением, может быть, тех областей, где особое значение придается надежности и безопасности. Почему я так думаю? Потому что убежден в том, что реальный мир никогда не сможет быть описан единственной формальной теорией.

3.6.3. Объектно-ориентированное программирование при помощи обычных языков

Поскольку разработка программного обеспечения должна быть организована в соответствии с некоторой технологией объектно-ориентированного анализа и проектирования, рассмотренной в главах 6—9, многие пользователи, обеспокоенные низкой производительностью и сложностью изучения объектно-ориентированных языков, могут рассматривать
возможность реализации своих систем при помощи обычных языков. Большая часть пуристов (борцов за "чистоту нравов" — *примеч. пер.*) в этой области со злостью говорит о такой альтернативе, которая все больше и больше становится ненужной. Однако мы увидим, можно ли это осуществить на практике.

Хотя и существуют специальные объектно-ориентированные (по крайней мере, основанные на объектах) версии языка Pascal, реализовать объектно-ориентированную спецификацию при помощи обычного языка Pascal очень сложно, главным образом, из-за главного его преимущества — блочной природы. В таких структурах основное внимание уделяется процессу декомпозиции; и хотя в нем представлены абстрактные свойства, например тип перечисления, все управляющие структуры являются, конечно, функциональными по своей природе.

Реализация объектно-ориентированного проектного решения при помощи языка COBOL представляет собой, в лучшем случае, странное зрелище. Замечания, сделанные по поводу языка Pascal, в некоторой степени справедливы и для языка COBOL.

С помощью языков FORTRAN и C действительно можно реализовать объектно-ориентированные проектные решения, и в самом деле существует теорема, согласно которой язык Objective-C, в известном смысле, реализуется при помощи языка C. Однако реализовать это сложно и дорого. В языке FORTRAN отсутствие динамических массивов требует определенных расширений, а некоторые другие программные конструкции этого языка усложнили бы выполнение подобных разработок.

Если настоящие программисты пишут на языке FORTRAN, то *действительно* крутые парни пишут на языке ассемблера. Было даже сделано несколько попыток написать язык ассемблера в объектно-ориентированном стиле. В [232] даются некоторые указания по поводу осуществления этой идеи.

В языках Ada и Modula-2 объектно-ориентированное программирование доступно только отчасти, при значительных усилиях, и, как правило, в этом случае необходима дополнительная документация. Эти языки уже были рассмотрены и признаны скорее основанными на

объектах, а не объектно-ориентированными языками, и, конечно, для них характерны аспекты инкапсуляции, присущие объектно-ориентированному подходу. Универсальность может использоваться для моделирования некоторой ограниченной формы наследования, однако отсутствует естественный способ его развития.

В [674] представлен ряд полезных эвристик для реализации объектно-ориентированного проектного решения при помощи обычного языка программирования. Отредактированная версия этих рекомендаций (которые я нахожу весьма полезными) выглядит следующим образом.

- Классы преобразуются в следующие структуры данных: структуры — в языке C, записи и пакеты — в языке Ada; массивы и общие блоки — в языке FORTRAN; записи — в языке Pascal.
- Сообщения преобразуются в обращения к функциям.
- Память выделяется для объектов с целью хранения глобальных и стековых переменных.
- Иерархии наследования "становятся более плоскими" посредством преобразования каждого неабстрактного класса в структуру и повторной реализации унаследованных методов каждого такого класса. Обычно это делается посредством вызова процедур.
- Полиморфные ссылки разрешаются на стадии компиляции посредством идентификации класса каждого объекта и на стадии выполнения — посредством проверки каждого экземпляра или использования команд `select/case`.
- Ассоциации, относящиеся к модели данных, преобразуются в указатели или реализуются непосредственно как структуры данных.
- В целях сохранения инкапсуляции следует избегать глобальных переменных, область видимости должна быть ограниченной и для доступа к полям различных классов следует использовать методы доступа. Прямой доступ к ним должен быть запрещен.

Надо признать существование значительных инвестиций, вложенных в программы, написанные на обычных языках, например на языке COBOL. Защитить эти инвестиции во время перехода к объектно-ориентированному программированию поможет важное понятие *объектной оболочки* [230]. *Объектные оболочки* могут создаваться для существующего кода, который впоследствии может быть постепенно заменен. Объектная оболочка обеспечивает взаимодействие новой, объектно-ориентированной части системы с более традиционной частью системы посредством передачи сообщений (рассмотрение вопросов, связанных с объектными оболочками, содержится в главе 4).

Многие специалисты, особенно в области объектно-ориентированного программирования, считают, что реализация объектно-ориентированного проектного решения при помощи обычного языка невозможна или, по крайней мере, нецелесообразна. Как я уже упоминал, здесь существует аргумент в виде языка Objective-C. Ясно, что такая реализация осуществима, но вопрос касается не возможности, а производительности. В конце концов мы можем выполнить объектно-ориентированное программирование даже при помощи машинного кода, поскольку это код, которым заканчивается разработка всех систем, даже интерпретируемых. Практическая проблема заключается в том, насколько легко создать и сопровождать такой код.

3.7. Выбор объектно-ориентированного языка

Каким образом осуществлять выбор инструмента или языка, который предоставит те преимущества, которые мы надеемся получить от использования объектно-ориентированных методов? Ответ не так прост. Во-первых, необходимо рассмотреть свойства и требования каждого приложения и попытаться выбрать инструмент, наилучшим образом подходящий для данной работы, поскольку объектно-ориентированная технология — это не панацея. Например, выбирая между языками C++ и Smalltalk, можно было бы оценить, являются ли структуры классификации, выявленные на стадии анализа, широкими и плоскими или узкими и глубокими. Во-вторых, мы серьезно должны подумать об организационных последствиях выбора: будет ли наша система соответствовать стандартам других систем или подходит ли она для имеющегося персонала? Ключевые мотивы в пользу применения таких языков, как Ada или C++, основываются именно на последнем утверждении. В качестве альтернативного варианта можно немного отступить от проблемы и задать себе вопрос: может, преимуществ легче достичь посредством использования объектно-ориентированного анализа и проектирования, откладывая выбор языка программирования на более позднее время или даже реализуя проект при помощи языков COBOL, FORTRAN или некоторых языков программирования четвертого поколения.

В следующей главе мы увидим, что создать сложные графические пользовательские интерфейсы практически невозможно без использования языка, поддерживающего высокий уровень многократного использования и масштабирования. Таким образом, организации, намеревающиеся создать графические пользовательские интерфейсы, должны серьезно подойти к вопросу перехода на объектно-ориентированный язык программирования. Аналогичным образом, если приложение предназначено для моделирования, то настоятельно рекомендуем решить вопрос в пользу объектно-ориентированного программирования.

Выбор конкретного языка или нескольких языков определяется при рассмотрении всех известных свойств приложения и прогнозируемых свойств, которые могут понадобиться в будущем. Затем можно выяснить свойства языка. Сложность такого подхода заключается в том, что почти невозможно гарантировать точность и полноту подобного прогнозирования. И чтобы застраховаться от будущих изменений, приходится искать язык, обладающий максимальным количеством требуемых свойств. Этот вопрос был бы достаточно простым, если бы существовали языки со *всеми* нужными свойствами или если бы одни свойства не противоречили другим. Например, по поводу максимального количества свойств можно сказать, что не существует объектно-ориентированного языка, обладающего как особенностями полной абстракции данных, так и абсолютно гибким множественным наследованием. Даже диалекты языка Smalltalk, реально поддерживающие множественное наследование, не обладают гибкостью упомянутых выше систем, ведущих свое происхождение от разработок в области искусственного интеллекта. Что же касается противоречащих друг другу свойств, то мы уже видели, что не существует "полностью объектно-ориентированного" языка, способного продемонстрировать высокое качество функционирования в большом объеме, при наличии значительного числа пользователей и в реальном масштабе времени.

Я рекомендовал бы специалистам вначале приобрести опыт, создавая прототипы на языке типа Eiffel, а затем, если разработки ведутся на рабочих станциях, перейти к разработке приложения на языке типа C++. Если требуется реализация базовой системы, я обычно советую им подождать и посмотреть. В следующем разделе я выскажу свою точку зрения по поводу того, чего же следует ожидать.

Любой язык, выбранный сейчас, рискует быть замещенным другим, пока еще несуществующим языком, который может стать промышленным стандартом. Здесь мы опять сталкиваемся с головоломкой. Следует ли ожидать эту зрелую технологию и затем приступать к ее изучению или лучше погрузиться в теперешнюю, зная, что некоторые результаты могут устареть. Я думаю, что компании, вступающие в эту сферу деятельности, должны основываться на том, что изучаемые ими технологии будут более ценными, чем те программы, которые они пишут. В этом смысле выбор несовершенного языка может не иметь большого значения. Удачное объектно-ориентированное модельное представление может оказаться более важным, чем окончательный выбор языка. Философия этой книги такова, что анализ и проектирование, выполненные с учетом многократного использования и наращивания, будут иметь долговременную значимость и позволят пользователям объектно-ориентированной модели воспользоваться преимуществами будущих языков, когда они появятся.

В последующих главах мы обсудим эти вопросы более подробно, рассматривая объектно-ориентированный анализ и проектирование как самостоятельную проблему, не связанную с реализацией, и пойдем в своих исследованиях еще дальше — обратимся к объектно-ориентированным базам данных. В следующей главе мы, например, приведем обзор некоторых приложений, типичных для систем объектно-ориентированного программирования, и попытаемся определить, каким образом можно выразить требования к приложению при помощи соответствующего языка.

3.8. Направления и тенденции

Существует два хорошо известных недостатка, препятствующих развитию объектно-ориентированных систем: они не допускают постоянного масштабирования и не достаточно эффективны. Опыт показывает, что в рамках этого подхода могут быть созданы большие системы. Эти приложения функционируют в различных областях и включают управление сопровождением, системы управления обучением в реальном масштабе времени, системы, обеспечивающие принятие решений, финансово-торговые системы, системы диагностики аппаратного обеспечения, программные продукты для автоматизированного проектирования и географические информационные системы, системы для отслеживания перемещений контейнеров на погрузочной платформе и сканеры компьютерной томографии CAT[4]. Последние представляют собой интересный пример объектно-ориентированных систем, предъявляющих особые требования к обеспечению безопасности. Что касается обычных систем, то здесь, как обнаружилось, нужны качественные изменения примерно 100000 строк кода, где все оказывается сложнее и абсолютно несоизмеримо с небольшим дополнительным расширением функциональности. Опыт показывает, что с объектно-ориентированными системами этого просто не может произойти. Хотя этот факт не свидетельствует о серьезности упомянутых выше недостатков, но подобные достижения характерны только для приложений определенных типов в рамках конкретного диапазона. Реальный вопрос состоит не в том, является ли объектно-ориентированное программирование неэффективным или масштабируемым, а в том, какие приложения могут создаваться с его помощью. Ответ на этот вопрос может дать только опыт, хотя, как мы уже видели, подобный опыт будет сопровождаться

[4] CAT — сканер, предназначенный для компьютерной томографии; это разновидность рентгеновского сканера, который при работе фокусирует лучи на нескольких параллельных плоскостях.

появлением новых языков, обладающих новыми свойствами, а также новыми библиотеками классов и каркасами приложений.

Следует признать, что некоторые системы реального времени действительно сталкиваются с неразрешимыми проблемами производительности. Создание быстрых систем реального времени возможно, но только в определенных прикладных областях.

Библиотеки должны тестироваться на базе реальных приложений. Некоторые специалисты полагают, что в целях поддержки многократного использования компоненты должны поставляться в виде двоичного кода. Особые проблемы тестирования связаны с наследованием, поскольку наследование является причиной побочных эффектов и может заставить класс реализовать переменные экземпляров его предка. Средства тестирования должны автоматически выполнять тестирование на всех уровнях наследования. Такой инструмент тестирования быстро оправдает свою стоимость даже в совсем небольших проектах. Стратегию тестирования можно обобщить при помощи следующего утверждения: "прежде всего следует попытаться устранить наихудшую ошибку". Специалисты по тестированию должны задавать себе вопрос: "Если здесь появилась такая ошибка, где она может появиться еще?" Объектно-ориентированное программирование продемонстрировало, что при надлежащем управлении оно позволит снизить показатель частоты появления ошибок — почти вдвое. Как правило, объектно-ориентированные системы действительно оказываются более надежными.

Современный опыт продемонстрировал еще и то, что менеджеры проектов на самом деле нуждаются в реализации идеи, предполагающей необходимость надлежащего обучения программистов, пишущих на объектно-ориентированных языках. Чтобы почувствовать себя комфортно, работая в объектно-ориентированном стиле, потребуется примерно шесть месяцев. Наилучший способ обучения — выполнять работу совместно с опытными специалистами, выступая в роли подмастерья. Обычная ошибка начинающих состоит в том, что при конструировании объектов они слишком увлекаются разбиением их на очень мелкие детали.

Реализация должна быть такова, чтобы жизненный цикл разработки объектно-ориентированных систем принципиально отличался от принципов разработки традиционных приложений. Объектно-ориентированные проектные решения больше похожи на математическое моделирование, чем на традиционные проектные решения бизнес-систем, и вообще выглядят скорее как имитационные модели, а не как плоды анализа и проектирования. Для проектов такого рода модель водопада не используется, но имеет важное значение создание прототипов. В целях многократного использования программного обеспечения должна быть проведена техническая экспертиза каждой отдельной строки кода. Отчасти это объясняется тем, что любому программисту очень сложно представить все, что может находиться в библиотеках кода, а также тем, что работа в составе команды значительно облегчает выполнение: кто-нибудь другой может знать, что "именно тот класс, который вам нужен", есть в такой-то библиотеке.

К двум основным препятствиям, тормозящим распространение объектно-ориентированного стиля, можно отнести проблемы, связанные с производительностью, и привычку применять традиционные модели жизненного цикла и подходы к проектированию.

Между тем менеджеры проектов, которые намереваются осуществить переход к объектно-ориентированному программированию, должны быть правильно информированы насчет создания объектных оболочек для существующего кода, который впоследствии может быть заменен или удален. Привлеченные перспективой роста производительности в соотношении 24:1, большие компании отстаивают сейчас эту идею. По мере неуклонного совершенствования аппаратного обеспечения становятся модными языки, подобные языку Java. Я знаю один проект, который перешел с языка Pascal на Smalltalk, и система действительно заработала

быстрее, поскольку в объектно-ориентированной среде можно достичь большего понимания системы в целом, что позволяет упростить выполнение глобальной оптимизации.

Похоже, что к настоящему моменту C++ и Java стали наиболее успешными, наиболее действующими и универсальными объектно-ориентированными языками программирования.

Объектно-ориентированные языки программирования являются мощным, но недостаточно зрелым инструментом, поскольку они быстро приобретают новые качества; кроме того, все еще появляются новые языки. Для некоторых приложений, например, разработки графических пользовательских интерфейсов и распределенной обработки данных они абсолютно незаменимы, о чем и расскажет следующая глава.

3.9. Резюме

Существует огромное множество языков, связанных с идеей объектно-ориентированного подхода. Их диапазон простирается от "чисто" объектно-ориентированных языков программирования (типа языка Smalltalk) до языков, основанных на объектах (например, язык Visual Basic), но их строгая классификация отсутствует. Я провел следующую специальную классификацию, хотя это и не претендует на то, чтобы быть рассмотренным в качестве исчерпывающего списка.

- Чисто объектно-ориентированные языки

  CLOS
  Component Pascal
  Eiffel
  Java
  Simula
  Smalltalk
  Prolog++ и DLP

- Расширения обычных языков или гибридные объектно-ориентированные языки

  C++
  Objective-C
  Object Pascal
  Modula-3 и Ada 95
  Object COBOL

- Расширения языка Lisp и других систем искусственного интеллекта

  KEE, Joshua и ART
  KBMS и ADS
  Nexpert Object и Level 5 Object
  ProKappa, Kappa, ObjectIQ и XShell

- Языки, основанные на объектах

  Ellie
  Modula-2

PowerBuilder
Visual Basic

- Языки, основанные на классах

  CLU

Объектно-ориентированные проектные решения могут быть реализованы и при помощи обычных языков, но сделать это сложно, и многие преимущества могут быть потеряны.

Для перехода к объектно-ориентированному программированию и защиты инвестиций, вложенных в обычный код, могут быть использованы объектные оболочки.

Объектно-ориентированный подход — это модельное представление, не имеющее в своей основе формальной теории, однако в этом направлении ведутся научно-исследовательские работы.

Каскадный жизненный цикл не подходит для объектно-ориентированного программирования. Важное значение приобретают итерации, а при использовании библиотек классов крайне желательна работа в команде. Этот вопрос рассматривается далее в главе 9.

Вплоть до настоящего времени казалось, что C++ станет самым успешным, самым используемым и универсальным объектно-ориентированным языком программирования, по крайней мере, для рабочих станций и приложений небольших масштабов. Однако все больше и больше проявляется интерес к языку Java. Очень большие надежды в отношении будущего объектно-ориентированного программирования в сфере универсальных вычислительных машин и коммерческих систем возлагались на язык Object COBOL, однако он не стал популярным.

Объектно-ориентированные языки программирования являются очень мощными, но все еще незрелыми языками в том смысле, что они быстро приобретают новые свойства, и, кроме того, все еще появляются новые языки. Для многих приложений они являются достаточно зрелыми с точки зрения их практического применения, а в некоторых случаях — просто незаменимыми. Для других приложений необходимо разрабатывать библиотеки классов как часть флагманских проектов для крупных пользователей.

3.10. Дополнительная литература

Первоисточником по языку Simula является работа [207], но к настоящему времени появились некоторые нововведения, и часть из них описана в [451].

Существует множество вводных учебников по языку Smalltalk, включая конструктивную книгу [302], и много других более поздних публикаций, которые ввиду их многочисленности не могут быть здесь перечислены. В качестве простого учебника я предпочитаю книгу [339]. Одними из наиболее полных руководств по языку Smalltalk 80 являются работы [465, 466]. Для приверженцев продуктов Macintosh в [687] содержатся практические инструкции по использованию языков Object Pascal и MacApp [37] и соответствующих библиотек классов. Еще одна превосходная трактовка представлена в книге [491].

В [733] приводится подробное описание оригинального языка C++, а также философское обоснование этого подхода. Более поздняя редакция этой книги вышла в 1997 году [736]. Монография *Design and Evolution C++*, изданная в 1994 году, также имела важное значение. Другие учебники для начинающих — [246, 254, 358, 489, 577, 630, 633, 781].

Вместе с руководством [221] предлагается компакт-диск с учебными материалами. Книга [796] — хорошее популярное издание, уделяющее особое внимание новым свойствам языка C++, например использованию шаблонов. Необычно то, что при этом не требуется знания языка C. В [579] описана стандартная библиотека шаблонов STL (Standard Template Library). В книге [634] дается краткое описание международного стандарта ANSI (American National Standards Institute — Американского национального института стандартов) для языка C++. Руководство [191] хорошо известно как самая лучшая книга для опытных программистов, пишущих на языке C++. В [198] содержится общий вводный курс в объектно-ориентированное программирование, щедро проиллюстрированный примерами, написанными на языке Objective-C, и сравнениями с другими языками.

Самыми лучшими публикациями о языке Eiffel несомненно являются книги [547, 549], рассматриваемые в последней главе, где этот язык снова сравнивается с другими языками. Мейеру также есть что рассказать о трудностях объектно-ориентированного программирования, реализуемого при помощи обычных языков.

Языку Java посвящено бесчисленное множество книг, и эталоном среди них является монография [310]. К другим важным публикациям можно отнести [274, 482, 797] и мою любимую книгу [128]. В работе [537] подробно рассматривается модель безопасности Java.

Принципы, на которых базируется язык Object-COBOL, рассматриваются в [68]. Книги [39, 760] посвящены более глубокому исследованию языка. Книга [531] посвящена объектно-ориентированному расширению языка Prolog и логическому программированию в целом. В [253] описан исследовательский язык DLP, намного более мощный, чем язык Prolog++, который также призван решать проблемы параллелизма. Современный объектно-ориентированный язык Ellie представлен в работе [31]. В [242] дано краткое описание языков BETA и MjШlner.

В [98] приводятся подробные примеры действующих объектно-ориентированных систем, целиком реализованных на языках C++, CLOS, Smalltalk, Object Pascal и Ada. Во втором издании данной книги эти примеры отсутствуют.

В [752] рассматриваются языки, связанные с разработками в области искусственного интеллекта, — Nexpert Object, Goldworks, ART, KEE, LOOPS, CLOS; в настоящее время они являются весьма устаревшими.

В работах [759, 814] описаны альтернативные объектно-ориентированные языки и языки на основе исполнителей. В работе [546] уделено внимание вопросам параллелизма и объектно-ориентированного программирования.

В [712] затронуты некоторые менее известные аспекты теории объектно-ориентированных языков программирования. Статья [141] представляет собой классический труд по вопросам полиморфизма. В этой статье содержится также описание исследовательского языка FUN, упомянутого в разделе 3.3.

В замечательной работе [569] представлены идеи, лежащие в основе CLOS (системы объектно-ориентированного программирования на языке Common Lisp), а работа [720] вводит в мир языка Common Objects. Обе статьи дают ценные рекомендации по использованию наследования без ущерба для абстракции и ее основного преимущества — многократного использования.

В статье [210] можно найти прекрасный и глубокий обзор исследований в области теорий типов для объектно-ориентированных языков программирования, а также многие интересные комментарии общего характера, относящиеся к объектно-ориентированному подходу. В этой работе можно найти также дополнительные ссылки на языки, упомянутые в разделе 3.6.5. Более поздняя работа [1] тоже посвящена этой теме.

Объектно-ориентированная машина Rekursive описана в работе [360] для предварительного ознакомления.

Публикация *Object lessons* [497] изобилует информацией, касающейся практического опыта и делового прагматизма в отношении объектно-ориентированного программирования и созданных на его основе приложений.

В *Journal of Object-Oriented Programming* можно найти самые новые полезные материалы; а более специальная информация, касающаяся соответствующих языков программирования, представлена в журналах *C++ Report, The Java Report* и *The Smalltalk Report*.

3.11. Упражнения

1. С какими языками программирования связывают зарождение объектно-ориентированного программирования?
   - **а)** Algol 68
   - **б)** PL/1
   - **в)** Simula
   - **г)** Pascal
   - **д)** Ada
   - **е)** Lisp
   - **ж)** заменяемость
2. Кто из приведенных ниже специалистов *не* участвовал в разработке языка Smalltalk?
   - **а)** Адель Голдберг
   - **б)** Алан Кэй
   - **в)** Бьярн Страуструп
   - **г)** Дэн Ингаллс
   - **д)** Брэд Кокс
3. Сравните свойства двух известных вам объектно-ориентированных языков.
4. Почему язык Java так быстро достиг успеха, а язык Eiffel — нет?
5. Почему главные преимущества и просчеты связаны с языком C++?
6. Укажите различия между Java Beans и Enterprise Java Beans.

4

Распределенные вычисления, программы среднего уровня и перенос систем на новую платформу

От каждого — по способностям,
каждому — по потребностям.

К. Маркс. *Критика Готской программы*

Темой настоящей главы является совместное использование данных и функций разными системами, платформами и процессами. Это приводит нас к концепциям распределенных вычислений, брокеров объектных запросов, архитектуры **клиент/сервер**, программ среднего уровня, компонентно-ориентированной разработки CBD (Component-Based Development) и интеграции приложений в рамках предприятия EAI (enterprise application integration). Распределенные системы состоят из узлов, которые с помощью своих интерфейсов предоставляют услуги другим узлам по мере поступления запросов. Эта глава сконцентрирует внимание читателя на практических вопросах перехода от централизованных (обычных) систем к распределенным. При этом некоторые компоненты распределенных систем останутся прежними, а остальные будут строиться с помощью объектно-ориентированного программирования. Прежде всего, хотелось бы подчеркнуть роль объектно-ориентированной модели в описании и понимании распределенных систем. Многие современные компании уже определились, что по возможности в своих будущих системах будут использовать

распределенную архитектуру. Следовательно, им необходимо выработать стратегию переноса (миграции) существующих систем на новую платформу. Если среди их аппаратных средств имеются большие машины, на них будут выполняться специализированные приложения, предъявляющие высокие требования к обработке транзакций. Кроме того, большие машины могут использоваться как серверы данных, если существующая корпоративная база данных не может быть заменена более экономичной. Успешное смещение в сторону электронной коммерции, основанной на Web-технологиях, означает, что существующие системы должны быть интегрированы с современными объектно-ориентированными программами. Многие связи можно организовать с помощью технологии XML.

В некотором смысле распределенные вычисления — это переход от хаоса (или свободы) персональных компьютеров обратно к золотому (или черному) веку больших машин. Первые компьютеры были однопользовательскими и обладали малым объемом дискового пространства для хранения данных. Операционные системы с разделением времени обеспечили возможность совместного использования одной и той же архитектуры несколькими пользователями. Когда появились глобальные сети (WAN — wide area network), такая кооперация развернулась в масштабах всей планеты, сохраняя при этом несколько центральных точек управления интеграцией и администрирования. Появившиеся затем рабочие станции повернули процесс вспять к однопользовательским машинам, которые "варились в собственном соку" и были изолированы друг от друга, несмотря на постоянное стремление пользователей к кооперации и совместному использованию ресурсов. Локальные сети и Web положили конец этому карантину, однако вернули сложность работы с большими машинами. Только теперь большие машины стали распределенными, а управление ими стало сложным, как никогда. Тем не менее в сумме выигрыш перевесил повышение сложности. В принципе, распределенные вычисления оказались настолько более быстрым, гибким, масштабируемым и открытым механизмом, что способны поддерживать даже глобальные и распределенные производства.

В этой главе будут рассмотрены основные идеи, положенные в основу распределенных систем, и изложены базовые идеи распределенных объектных вычислений. После этого будут описаны концепции брокеров объектных запросов и ориентированных на сообщения программ среднего уровня. В заключение мы рассмотрим вопросы построения стратегии миграции. В главе 10 эти вопросы будут пересмотрены еще раз, но уже с позиций электронной коммерции.

4.1. Распределенные вычисления и архитектура клиент/сервер

Распределенные системы можно представить как группу объединенных в сеть компьютеров, которые совместно не используют оперативную память (как это делают многопроцессорные компьютеры). Это значит, что узлы должны обмениваться информацией с помощью сообщений, а это наводит на мысль об использовании объектно-ориентированной модели для построения таких систем. К тому же объекты служат подходящей парадигмой для объединения данных и элементов управления, а также естественной единицей распределения. Повышение производительности достигается за счет неявного параллелизма. Дорогостоящие и недостаточно загруженные ресурсы, такие как графопостроители, могут для экономии средств использоваться совместно. В грамотно построенных системах не должно существовать одной

точки отказа (single point failure) или должна быть возможность дублирования служб с целью повышения доступности данных и надежности системы. При добавлении узлов система должна реконфигурироваться по частям, что сокращает затраты и облегчает модернизацию (примерно как в высококлассных аудиосистемах). Эти и другие особенности должны быть оценены с учетом накладных расходов, связанных с обслуживанием сложной архитектуры, и трудностей, связанных с ее пониманием и описанием. Следует заметить, что компактные, простые системы гораздо проще настраивать и использовать, чем системы, составленные из многих компонентов, хотя их мощность, качество и гибкость — на порядок ниже.

Типы распределенных систем

Операционные системы распределенных систем могут быть **распределенными** (distributed) и **сетевыми** (networked) [747]. В первом случае операционная система сама распределена по всем узлам, что для пользователей практически скрывает наличие сети. В сетевых операционных системах на каждом узле содержится полная копия системы, при этом сама сеть становится видимой.

В распределенных вычислениях можно выделить три стратегии управления данными: централизованная, на основе репликации и разделения. **Централизованное** (centralized) управление предполагает размещение данных на одном узле, куда будут маршрутизироваться все запросы. Особенностью этой стратегии является то, что изменения в данных нужно выполнить только один раз. Недостаток в том, что результат запроса размещается в сообщении и передается по сети. При **реплицированном** (replicated) управлении копии данных создаются там, где в них наиболее часто возникает потребность. Это позволяет избежать двухэтапного подтверждения[1], но может привести к тому, что пользователь будет работать с устаревшими данными, если не будут предприняты дополнительные сложные меры. Учитывая, что доступ для чтения запрашивается гораздо чаще, чем доступ для записи, такая стратегия не так уж плоха. Стратегия **разделенного** (partitioned) управления предполагает распределение данных по узлам, на основе вероятности запросов к ним. Она требует двухэтапного подтверждения, но может оказаться эффективной в том случае, когда распределение основывается на естественных законах принадлежности данных. Объектно-ориентированная декомпозиция может оказаться даже лучше, так как она строится с учетом не только отношения принадлежности, но и концепции стабильности. На сегодня производители систем управления базами данных в качестве стратегий распределения данных предлагают выбор между двухэтапным подтверждением и репликацией.

Отдельно от типов операционных систем и способов управления данными стоят различные типы распределенных архитектур компьютеров, и среди них легко запутаться. Архитектура **клиент/сервер** (client-server) предполагает наличие одного сервера и одного или нескольких клиентов, при этом основной объем работы выполняется на компьютерах клиентов. Часто сервер выступает просто хранилищем файлов или баз данных, хотя такая же терминология используется и по отношению к серверам печати. Это — достаточно простая форма распределения, в которой интерфейс пользователя и бизнес-логика обычно перемешаны. **Трехуровневая** (three-tier) архитектура клиент/сервер предполагает выделение интерфейса

---

[1] Двухэтапное подтверждение обеспечивает успешное завершение транзакции на всех участвующих машинах только тогда, когда все они подтвердили успешное завершение операции обновления данных, связанной с транзакцией. Если какой-либо узел не ответил положительно (или задержался с ответом) после первого этапа обновления, все узлы возвращаются к состоянию, предшествовавшему обновлению.

пользователя и бизнес-логики в отдельные уровни. **Многоуровневая** (N-tier) архитектура может сделать это разбиение еще более подробным. Общим решением для Web-приложений является архитектура на основе так называемых **тонкого** (thin) и **толстого** (thick) клиентов, предполагающая вынесение некоторых элементов бизнес-логики в клиентскую часть приложения (примером может служить Java-аплет, выполняемый Web-броузером). На следующем уровне сложности находится **мультиклиентно/мультисерверная** (multiclient/multiserver) архитектура. В ней может существовать несколько серверов, однако они не взаимодействуют напрямую. Другими словами, узлы не могут совмещать в себе роль клиента и сервера. Такую архитектуру поддерживают некоторые базы данных. Однако самым общим является случай, когда узел совмещает в себе функции клиента и сервера (одновременно), хотя такую архитектуру сегодня можно встретить не часто. Такая архитектура носит название **сети равноправных абонентов** или соединения "точка-точка" (peer-to-peer). Узлы могут представлять собой процессоры, а также образы задач в процессоре. Сообщения при перемещении от узла к узлу могут быть разбиты на части, направлены разными маршрутами и снова собраны. Внедрение таких систем имеет свои сложности.

Еще один способ представления различных типов архитектуры представлен на рис. 4.1. На нем показано пять моделей: модель сервера баз данных, сервера транзакций, сети равноправных абонентов, распределенная и многоуровневая модели. Модель сервера баз данных предлагает ограниченный выбор мест для размещения функций системы, в то время как модель сервера транзакций более эффективна для балансировки нагрузки между узлами и сокращения сетевых потоков. В этих системах задачи обработки больших объемов данных можно возложить на сервер, а операции обслуживания интерфейса пользователя и дополнительные вычисления — на клиента. К тому же SQL-запросы могут находиться и в скомпилированном виде (в виде хранимых процедур) для повышения производительности, и это становится главной точкой поддержки.

Модель сети равноправных абонентов является наиболее общей и гибкой и одновременно самой сложной для программирования и администрирования. В таких приложениях, как управление процессами реального времени, обязательно определение порядка работ и программного обеспечения рабочих групп. Распределенная модель является единственной, которая использует стратегию оболочки. В ней имеется некоторая большая машина с символьным интерфейсом, основанным на синхронном терминальном протоколе (например, 3270). Для передачи данных и получения результатов на клиентской рабочей станции осуществляется преобразование форматов данных.

Трехуровневая модель становится все более популярной, однако немного позже мы увидим, как она может быть заменена многоуровневой моделью и моделью сети равноправных абонентов, основанными на программном обеспечении среднего уровня, которое скрывает основные сложности этих моделей.

Модель клиент/сервер

Модель вычислений клиент/сервер (CSC — client-server computing) можно определить как разделение процедур обработки и данных между одной или несколькими клиентскими машинами, на которых запущены приложения, и единственным сервером, который осуществляет обслуживание всех клиентов. Обычно эти машины объединены компьютерной сетью, и на клиентских компьютерах реализован графический интерфейс пользователя. В общем, систему клиент/сервер можно определить как такую, в которой отдельные элементы вычислений (интерфейс пользователя и доступ к базе данных) выполняются независимыми приложениями

или службами на одной или нескольких машинах. В этом контексте все объектно-ориентированные системы являются системами типа клиент/сервер; обратное утверждение неверно.

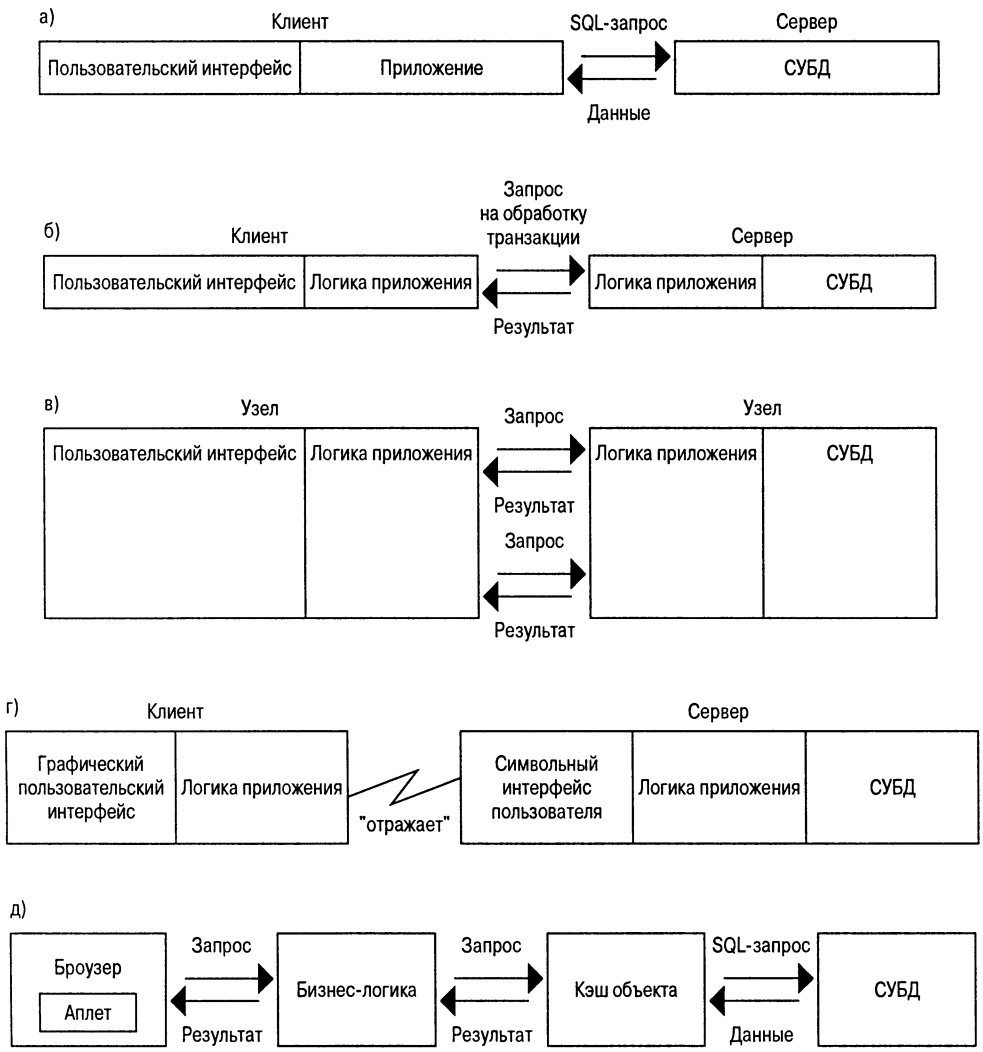


Рис. 4.1. Архитектуры распределенных вычислений: а) модель сервера баз данных; б) модель сервера транзакций; в) модель сети равноправных абонентов; г) распределенная модель; д) многоуровневая модель с тонким клиентом

Как мы видим, модель вычислений клиент/сервер представляет собой особый случай распределенных вычислений, в котором сама база не является распределенной (хотя может быть и таковой). Не следует путать эту модель с эмуляторами терминалов или графическими

программами переднего плана. К тому же CSC не является новой технологией. В 1970-х годах прошлого века терминалы обычно подключались к процессорам переднего плана (как правило, мини-компьютерам), которые, в свою очередь, подключались к приложениям, работающим на больших машинах (чаще всего — к базе данных или системе моделирования). В 1980-х годах для этого уже использовались персональные компьютеры, оснащенные эмуляторами терминала, и небольшой объем логики приложения, касающейся отображения на экране, переместился на эти компьютеры. Технология CSC открыла новые горизонты с появлением многозадачных операционных систем для персональных компьютеров. Теперь пользователь уже мог обслуживать соединение с удаленным сервером, одновременно выполняя локальные приложения.

Нельзя не сказать, что системы совместного доступа к ресурсам, такие как файл-серверы, не относятся к той же категории, что и серверы баз данных. Работая с файл-сервером, клиентское приложение запрашивает файл, который при этом блокируется и пересылается на обработку клиенту. Если доступ предоставлен для изменения, файл будет заблокирован до тех пор, пока клиент не завершит работу с ним. Вся обработка выполняется на компьютере клиента. С другой стороны, если на сервере запущена система управления базами данных, клиент пересылает запросы и результаты обновления данных. Сервер обрабатывает их и возвращает только результат. Серверная обработка используется совместно многими клиентами, потоки в сети сокращаются, при этом блокировка минимизируется.

Большая часть систем типа клиент/сервер используется при работе с серверами баз данных. Однако при этом выделяется множество уровней абстракции (рис. 4.2). Разделение функций между клиентом и сервером показано по горизонтали. Серверы приложений соответствуют объектам и оболочкам объектов. Давайте немного обобщим идею сервера баз данных до сервера объектов домена и проведем ее разграничение от сервера объектов приложений. С этой точки зрения на рисунке становится видна многоуровневая архитектура, а не просто архитектура клиент/сервер. Становится видно, что объектно-ориентированные системы — не что иное, как разложенные по уровням мультиклиентно/мультисерверные системы, или системы равноправных абонентов. На рис. 4.2 показано шесть уровней; реальное их количество зависит от самого приложения. Светло-серые области свидетельствуют о возможности введения дополнительных уровней.

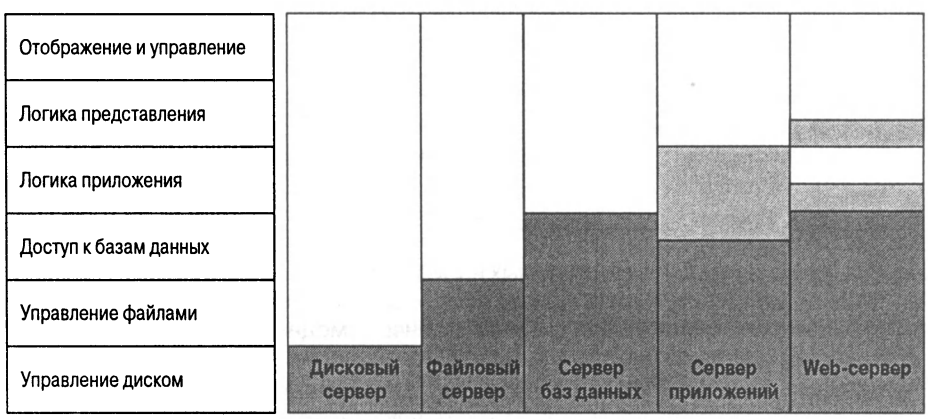


Рис. 4.2. Уровни абстракции систем клиент/сервер

Традиционные утилиты баз данных чаще всего организованы в рамках парадигмы клиент/сервер. Одним из первых коммерческих продуктов, в котором был реализован такой подход, была система управления базами данных Sybase; позже мультиклиентно/мультисерверная архитектура была реализована в СУБД Oracle и Ingress. Изначально подход к базам данных в Sybase предлагал каждому клиенту одну точку доступа к многопоточному серверу. Существовал единый процесс, в котором для каждого пользователя создавался отдельный поток. Такой подход эффективен с точки зрения использования памяти, однако не применим в современных многопроцессорных компьютерах, или в компьютерах с параллельными процессорами. Мультисерверный подход предоставил каждому клиенту собственный однопоточный процесс, и это позволило лучше использовать параллелизм, несмотря на требование немного большего объема оперативной памяти. Средства такого типа предполагают наличие сетевой поддержки и следование определенным стандартам. Приложения клиента и сервера взаимодействуют посредством программного интерфейса приложения API (например, в Sybase использовался интерфейс "Open Client"), программного обеспечения подключения к серверу (например, SQL Connect) и ряда сетевых протоколов (таких, как TCP/IP, LU6.2/APPC от IBM или SPX/IPX от Novell). Что касается реляционных продуктов, то через эти фильтры передавались, как правило, запросы SQL, однако это было не следствием объектной ориентированности баз данных или других типов приложений. Реляционные системы, использующие хранимые процедуры, позволяют уменьшить нагрузку на сеть. Это происходит потому, что транзакции не фрагментируются на ряд выражений и, к тому же, предварительно компилируются для еще большего повышения производительности. Естественно, объектно-ориентированные системы тоже обладают этим свойством за счет хранения методов в соответствующих объектах. Мультиклиентно/мультисерверные системы расширяют модель CSC и предоставляют пользователям гетерогенный доступ к множеству серверов. При этом конкретное расположение сервера знать не требуется.

Хранимые процедуры реляционных систем не реализуют истинную объектную модель. Они позволяют рассматривать сервер баз данных как единый большой объект, т.е. создают оболочку. Сложность состоит в том, что в этих системах не реализована инкапсуляция. Робастный подход к объектно-ориентированному анализу должен быть реализован с помощью стандартов, процедур и политики организации. Для недисциплинированных (или плохо обученных) программистов открывается возможность получать доступ к структуре базы данных непосредственно с помощью запроса SQL, а не использовать для доступа соответствующие хранимые процедуры. Единственным, хотя и редко используемым решением этой проблемы может стать закрытие всех привилегий доступа пользователей к базам данных, чтобы при доступе использовались только хранимые процедуры. Однако в этом случае теряется большинство преимуществ реляционных баз данных (выражаясь в терминах возможности выполнения ассоциативных запросов по требованию). Двухуровневая архитектура клиент/сервер сегодня рассматривается как слишком жесткая для условий современного производства.

Сравнение централизованной и распределенной моделей

Существует противоречие между потребностью во взаимодействии всех приложений и необходимостью разрешения доступа каждого подразделения только к тем ресурсам, которые его касаются. Наряду с этим, все более сложной задачей становится техническое обслуживание. С одной стороны, пользователи могут не рассматривать резервирование и обслуживание дисковой системы как часть своей работы, с другой, — ничто так не раздражает, как удаление централизованной службой по выходным всех мелких полезных файлов на жестком диске.

Единственно возможным решением является автоматизация задач технического обслуживания, однако при этом сохраняется опасность тоталитарного подхода.

Существуют и другие области, в которых распределенные системы создают дополнительные сложности. Настройкой и резервированием централизованных систем занимаются операторы и администраторы. Физическое распределение часто делает непрактичным централизованное резервирование и управление версиями программного обеспечения. Здесь оказываются полезными сетевые файловые серверы общего пользования, но и они не решают проблемы полностью. Истинно распределенная система подразумевает распределение самой операционной системы, что придает множеству систем вид единой среды с точки зрения пользователя и администратора. Такая система должна гарантировать отказоустойчивость и параллелизм. Это значит, что сбой какой-либо одной подсистемы не приведет к сбою всей системы в целом и всегда будут существовать взаимосвязанные элементы, содержащие как процессор, так и память. Первое требование подразумевает, что узлы системы должны использовать общее состояние, т.е. в текущем контексте объекты должны быть реплицированы.

Высокая стоимость этого простого решения может также повысить и сложность. Полносвязные сети просты, но дороги, так как подразумевают использование сложных маршрутизаторов и методов планирования заданий. В [575] эта точка зрения проиллюстрирована на примере сравнения одной железной дороги, соединяющей два города и имеющей по одному пути в каждом направлении, с другой, которая имеет всего один путь плюс развязку для расхождения составов. Этот пример также иллюстрирует концепцию **запаздывания** (latency) в сети, которая часто игнорируется конструкторами распределенных систем. Железная дорога, имеющая один путь (без развязки) длиной 120 км и разрешающая скорость движения поезда 60 км/ч, обеспечивает двухчасовое запаздывание, в то время как железная дорога с двумя путями дает запаздывание, близкое к нулю. Запаздывание в компьютерных сетях определяется временем, необходимым для вызова другого потока управления. Сети с большой величиной запаздывания обычно имеют проблемы с синхронизацией времени. Запаздывание — важная концепция в конструировании объектно-ориентированных систем, не важно — распределенных или нет. В этом контексте она связана с задержками, вызванными блокировкой посылок.

Первые апологеты распределенного подхода обнаружили сложность, из-за которой существующие методы структурирования не только внесли малый вклад, но и затормозили развитие во многих областях. Это было связано с тем, что методы функциональной декомпозиции и отделения данных от их обработки не имели средств описания взаимодействия между автономными объектами. Поэтому появилась потребность в углубленных методах объектно-ориентированного анализа, которые помогли бы описать создаваемые системы. При этом потребовались новые навыки работы с распределенными вычислениями и, что более важно, — с различными типами оборудования и программных платформ, которые предполагалось использовать. Развитие стандарта XML еще раз обозначило потребность в точной семантике интерфейса. Несмотря на всю его мощь, самого стандарта XML не достаточно для определения семантики.

Одной из самых главных проблем распределенных вычислений, по сравнению с централизованными, является повышение сложности программ. Поэтому возникла необходимость в стратегии управления или снижения этой сложности. Именно здесь могут помочь объектная технология, обеспеченная средствами моделирования, основанными на передаче сообщений; библиотеки классов, защищающие разработчиков от сложности API-интерфейсов; встроенные маршрутизаторы и распределители запросов к объектам, упрощающие управление размещением и взаимодействие между процессами.

Распределенные системы должны стать еще более сложными, чтобы устранить проблемы, которые не возникали при построении обычных систем. Если один процессор дает сбой, а другой остается действующим, приложение может завершиться в неопределенном состоянии. Это вызывает необходимость в возврате к предыдущему состоянию для части приложения (избирательно и гладко). Представьте себе, что вы отправили чек в регион, где отказал почтовый сервер. И все это после того, как вы списали деньги со своего расчетного счета, предполагая, что они поступят на счет получателя.

Проблема возникает и в том случае, когда два процессора одновременно пытаются получить доступ к одним и тем же данным, а сама сеть дает сбой. Многие из читателей сталкивались с ошибками сети, которые происходят гораздо чаще, чем сбои автономных машин, и приводят к необходимости существенного повышения частоты сохранения при выполнении таких задач, как работа с текстовым процессором. На самом деле выбор между независимыми узлами и единой сетевой операционной системой часто напоминает выбор между эгоцентричным индивидуализмом и хорошо спланированной системой кооперации. Каждая из систем имеет свои преимущества — по крайней мере, для тех, кто ее выбрал.

Большая часть коммерческих объектно-ориентированных баз данных обеспечивает такую поддержку, хотя их архитектура широко варьируется в рамках поддержки совместного использования хранилищ данных и взаимодействия между компонентами клиента и сервера. Особенность такого подхода в том, что он позволяет обслуживать целостную объектно-ориентированную модель всего приложения. Наряду с этим хранение и восстановление данных, а также управление транзакциями, параллельными операциями и версиями обслуживаются автоматически самой базой данных. Среди недостатков — потенциально высокая стоимость преобразования уже существующих данных и (или) приложений при переходе к более производительной системе за счет использования объектно-ориентированных языков программирования. Как отмечено в [264], при переносе данных (в терминах расширения существующих систем) всегда возникают сложности. Они объясняются тем, что существующие приложения часто используют собственные методы управления экземплярами либо в форме обычной файловой системы, либо в виде систем управления базами данных. Авторы сделали вывод, что брокеры объектных запросов (ORB — object request broker), как правило, являются лучшим решением, даже если они не могут обеспечить достаточную гибкость объектно-ориентированного подхода. Они позволяют использовать существующие приложения, пакеты и другие ресурсы в рамках объектно-ориентированного каркаса. Однако многократное использование таких ресурсов имеет свойство обоюдоострой бритвы и иногда приводит к нарушению принципа небольшого интерфейса. Кроме того, брокеры объектных запросов строго не определяют, где и как приложения будут интегрироваться с уже существующими системами.

Прозрачность размещения

Объектно-ориентированная технология обеспечивает естественный метод моделирования распределенных приложений, так как сама объектная модель состоит из набора независимых элементов, каждый из которых имеет свои потоки управления и взаимодействия с другими объектами в форме передачи сообщений. В обычных системах интерфейсы библиотек построены в виде процедур. В распределенных системах все по-другому. Подобно тому, как от глаз пользователя скрывается состояние объекта, скрывается и его местонахождение (как часть состояния объекта). Однако существует ряд механизмов поиска таких объектов. Когда один объект ссылается на другой (например, передает ему сообщение), маршрутизация

сообщения не зависит от отправителя, даже если получатель находится на том же компьютере. Это характерно для всех объектно-ориентированных систем, так как объекты имеют свой уникальный жизненный цикл, независимо от их местонахождения. Другими словами, объектно-ориентированная модель предполагает и обеспечивает принцип **прозрачности размещения** (locational transparency).

За аксиому можно принять то, что распределенные вычисления всегда следуют принципу прозрачности размещения. Это значит, что пользователь никогда не заботится о физическом размещении служб, которые может запросить в любое время. Это верно с логической точки зрения, но задержки в сети иногда позволяют определить, что запросы выполняются удаленно. Возможны и более значительные расхождения, если доступ к службам обеспечивается посредством и локальной, и глобальной сети, так как их характеристики разительно отличаются. Вызовы удаленных процедур (RPC — remote procedure call) нарушают принцип прозрачности размещения, так как клиенту для вызова процедуры нужно знать, где она находится. Иногда это скрыто от глаз пользователя файловыми системами, такими как NFS от компании Sun, или сетевыми функциями, такими как DCE от OSF. Тем не менее проблема остается открытой. Работа таких служб зависит от конкретного типа оболочки, особенно программного интерфейса, и это представляет собой проблему. Немного позже мы покажем, как эту проблему можно обойти.

Среда распределенных вычислений (Distributed Computing Environment или DCE) представляет собой программную архитектуру, которая поддерживает распределение данных и совместное использование процессов. Она скрывает от пользователя применяемые протоколы связи, однако всегда основана на удаленном вызове процедур (RPC).

Понимание того, что в объектно-ориентированных системах, в принципе, возможна прозрачность местоположения, совсем не означает, что внедрение таких систем тривиально. На самом деле для поддержки такой среды требуются хорошие навыки программирования и знакомство с рядом технологий, в том числе и сетевых.

Преимущества

Главные преимущества распределенных вычислений состоят в следующем.

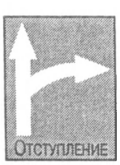
- Сокращение дополнительных затрат на удовлетворение запросов пользователей по обеспечению большей функциональности корпоративных систем.
- Уменьшение стоимости оборудования и разработки программного обеспечения за счет использования средств, оптимизированных для конкретных задач.
- Сокращение времени разработки за счет использования уже существующих систем в качестве готовых компонентов.
- Уменьшение затрат на аппаратные средства за счет использования дешевых рабочих станций и оптимизации использования прочего оборудования.
- Повышение конкурентоспособности клиентских приложений.
- Внедрение новых моделей бизнес-процессов, например, электронной коммерции.
- Облегчение интеграции за счет сокрытия сложности при обмене данными между машинами и привлечения новых стандартов.

Централизованные вычисления обладают преимуществами защищенности, экономичного масштабирования и простоты управления. Однако, с другой стороны, могут увеличиться операционные затраты, негибкость систем, стоимость разработки и обслуживания, а также может уменьшиться доступность систем и их управляемость. Кроме того, многие большие машины выполняют роль обычных калькуляторов, поэтому с их помощью трудно удовлетворить потребность пользователей в приложениях, моделирующих логику бизнес-процессов. Все это привело к настоятельной потребности в распределенных системах и вычислениях, ориентированных на конечного пользователя (end-user computing). Однако при этом остаются такие сложности, как низкая пропускная способность сети, возможность возникновения сбоев при подключении не всех компьютеров, затраты, связанные с недозагрузкой процессоров, бреши в системе безопасности, совместимость данных и задержки, связанные с блокировкой файлов. Во многих организациях появилась необходимость в привлечении новых специалистов из сферы информационных технологий, в переводе управления на качественно новый уровень и в повышении надежности. Командам специалистов по информационным технологиям потребовались не только старые знания, связанные с большими машинами, но и новые, относящиеся к персональным компьютерам, локальным сетям, новым сложным стандартам взаимодействия и компонентным моделям. По-моему, все только выиграли от понимания принципов объектных технологий и смогли на лету схватывать понятия объектно-ориентированного анализа. Естественно, это также потребовало дополнительных затрат (и часто довольно значительных) на привлечение кадров, получение консультаций и обучение.

Хотя распределенная архитектура всегда предполагала наличие двух или более машин, в принципе, как клиент, так и сервер могли физически размещаться на одном компьютере. Однако при реализации таких систем возникают как логические, так и физические различия. Изначально объектно-ориентированные языки программирования были ограничены только одним адресным пространством. Объекты, скомпилированные разными компиляторами (даже с одного языка), не могли взаимодействовать, поэтому библиотеки классов не могли поставляться в двоичной форме. Поэтому для решения проблем, связанных с межъязыковым и межобъектным взаимодействием, появились такие технологии, как RPC, COM и CORBA. Это привело к созданию компонентных моделей и метамоделей, таких как EJB и MOF.

Узлы сети можно рассматривать как абстрактные типы данных или объекты. Однако наследование и композиционные связи не распространяются на сеть. Ассоциации, охватывающие сеть, должны быть минимизированы из соображений эффективности. Как правило, системный уровень реализуется на одном узле. Остальные узлы (в лучшем случае) рассматриваются как оболочки компонентов.

4.1.1. Сетевые и архитектурные вопросы



Распределенные системы потребляют больше сетевых ресурсов, чем централизованные, и это нельзя недооценивать. Таким образом, стоит привлечь к дискуссии пользователей, которые, по крайней мере, знают, что они будут делать, специалистов по сетям и операционным системам и архитекторов приложений.

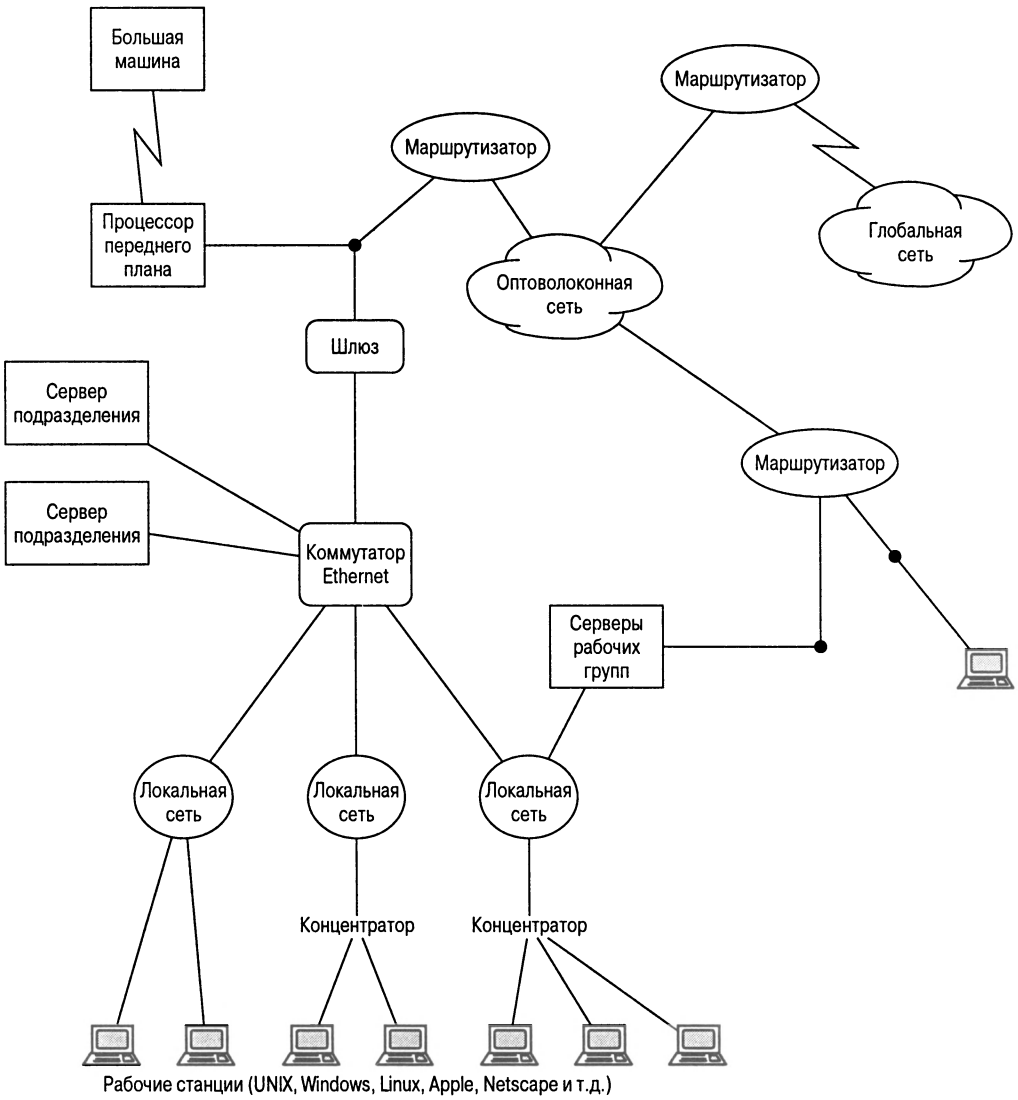


Рис. 4.3. Типичная архитектура сети в процессе перехода на распределенную архитектуру

Эти специалисты совместно способны построить обоснованный прогноз загрузки на сеть, позволяющий избежать ситуации, когда некто для переноса гигабайтовой базы данных по сети загружает ее на полчаса, в течение которых никто другой не может перенести свои данные.

Взаимодействие "точка-точка" может использоваться для строительства больших систем из относительно небольших компьютеров. Примером такой парадигмы является DECnet. Этой же модели соответствуют протоколы APPN от IBM и TCP/IP, хотя последний поддерживается более широко и является более открытым. Именно на этом протоколе построена

сеть World Wide Web. Один из наиболее опасных вопросов, касающихся перехода организаций к объектной технологии, состоит в разработке архитектуры системы, сети и приложений. Существенно то, что новые и уже существующие системы на протяжении переходного периода должны взаимодействовать. Часто "смесь" из разных машин, систем и сетей остается навсегда. На рис. 4.3 показана упрощенная версия типовой архитектуры переходного периода.

На этом рисунке на большой машине могут быть запущены все типы приложений COBOL, FORTRAN, DB2 и VSAM. При этом могут использоваться такие старые сетевые стандарты, как SNA. Ключевым компонентом, который нужно добавить к этой большой машине, является некая программа, позволяющая клиенту напрямую взаимодействовать с реляционными базами данных большой машины с помощью языка SQL или удаленно выполнять программы на языке COBOL. Еще одним ключевым компонентом является шлюз, представляющий собой специализированное программное обеспечение, запускаемое на выделенной машине; оно служит связующим звеном между клиентами UNIX и сетью больших машин. Некоторые производители баз данных также предлагают подобное программное обеспечение. Маршрутизаторы соединяют различные локальные и глобальные сети, а концентраторы обеспечивают соединения между рабочими станциями. Коммутаторы позволяют максимизировать гибкость и пропускную способность сети. Может потребоваться и другое оборудование, однако темой данной книги не является сетевая архитектура, поэтому упрощенной схемы, представленной здесь, вполне достаточно для общего понимания вопроса. В дальнейшем брокеры объектных запросов (object request broker) упростят эту картину, и эти вопросы будут рассмотрены в следующем разделе.

4.2. Брокеры объектных запросов и программы среднего уровня

Предметом нашего последующего рассмотрения будет прозрачное взаимодействие с существующими системами, пакетами и прочими объектно-ориентированными компонентами с помощью телекоммуникаций и локальных сетей. Задача состоит в том, чтобы приложения и службы были представлены в сети в рамках общей объектной схемы. Эта схема состоит из объектов, представляющих каждую из служб, а истинное размещение и реализация остаются прозрачными для пользователей и других объектов клиента. Специалисты группы Gather Group определили программное обеспечение среднего уровня (middleware) как "системы интерпретаторов, которые непосредственно реализуют взаимодействие между программами на прикладном уровне в среде распределенных вычислений". В то же время шутники дают другое определение программного обеспечения среднего уровня — это "программы, за которые никто не хочет платить". Другими словами, программы среднего уровня являются составной частью инфраструктуры систем распределенных вычислений. Существует несколько типов промежуточных программных продуктов, относящихся к различным уровням абстракции. На самом нижнем уровне находятся программы, основанные на передаче сообщений (MOM — message oriented software) (например, MQ Series от IBM или MSMQ от Microsoft), реализующие асинхронную маршрутизацию, сопровождение очередей и доставку сообщений. Программное обеспечение среднего уровня на основе передачи сообщений MOM представляет средний уровень в модели клиент/сервер. Это, естественно, не обеспечивает приемлемость формата сообщения для приложения-получателя, а значит не гарантирует, что

сообщение будет понято последним. По этой причине принято совмещать связующее программное обеспечение с семантическим уровнем, обычно основанным на XML.

Одним из распространенных способов взаимодействия приложений является стандартная архитектура брокеров объектных запросов CORBA (Common Object Request Broker Architecture). Все в большей степени эта архитектура объединяется с программным обеспечением среднего уровня на основе передачи сообщений MOM и мониторами обработки транзакций TP (Transaction Processing). Среди текущих примеров — интеграция компанией BEA Systems монитора Tuxedo TP с собственным продуктом ObjectBroker CORBA и с системой ETX от Tibco.

В 1989 году образовалась группа OMG (Object Management Group), объединившая влиятельные компании для выработки широкомасштабного соглашения между создателями терминологии объектно-ориентированной технологии и стандартов интерфейсов на основе уже существующей технологии. Изначально в OMG входили компании Borland, Microsoft, Hewlett-Packard, Data Central, AT&T, Unisys, Wang, ICL, Sun, DEC, а также множество других производителей компьютеров и программ и, кроме того, такие крупные пользователи, как Boeing и American Airlines. На момент написания этой книги в эту группу входило уже более 800 членов. Заседания технического комитета группы OMG происходили в Европе, Америке и на Дальнем Востоке, что подчеркивает ее интернациональную основу. Группа OMG организована для ускоренного выпуска стандартов и опережения более инертных государственных механизмов стандартизации. Она выпустила описание архитектуры и руководство по использованию стандарта CORBA для брокеров объектных запросов (ORB — Object Request Broker), независимую от языка компонентную модель, стандартную систему обозначений для объектно-ориентированного анализа и проектирования (язык UML), спецификации для стандартных бизнес-объектов вертикального рынка, метамодель для хранилищ данных (CWM) и Meta Object Facility (MOF). Некоторые поставщики предложили продукты, совместимые с CORBA. Быстрый рост группы OMG подтвердил, что промышленность уже созрела для принятия объектной технологии и требует стандартов.

Технология ORB — это прозрачная магистраль данных, соединяющая объектно-ориентированные приложения и модули переднего плана с существующими приложениями. В качестве аналогии можно привести стандарт коммуникаций электронной почты X500, позволяющий передать запрос другому приложению или узлу, даже не зная деталей структуры его службы каталогов. Таким же образом ORB практически устраняет потребность в сложных удаленных вызовах процедур, реализуя механизмы, с помощью которых объекты прозрачно создают и получают запросы и ответы. Целью этой технологии было обеспечение совместимости между приложениями на разных машинах в гетерогенной распределенной среде, а также связывания нескольких объектных систем. Брокеры объектных запросов обеспечивают средство абстрактного описания приложений и взаимосвязей между ними, а также работу служб обнаружения и использования этих приложений в гетерогенной сети. Приложения не обязательно должны быть написаны в объектно-ориентированном стиле, так как технология ORB предоставляет эффективную оболочку. Это вообще могут быть продукты сторонних производителей. Пакеты и настольные приложения могут быть многократно использованы и объединены для получения нового качества — распределенных, мультиплатформенных бизнес-систем. В [264] отмечено, что технология ORB привнесла преимущества объектных технологий в мир интеграции систем.

Брокеры объектных запросов — это продукты, основанные на программной архитектуре, разработанной группой OMG (рис. 4.4). В ней объекты делятся на объекты приложений (Application Objects), общие средства (Common Facilities) и объектные службы (Object

Services). Объекты приложений специфичны для конкретных приложений, таких как оболочки существующих систем, пакеты или электронные таблицы. Общие средства — это объекты, которые оказываются полезными в нескольких контекстах, например в справочной системе, сложных интерфейсах документов, броузерах, системах электронной почты и т.п. Объектные службы реализуют базовые операции логического моделирования и физического хранения объектов и могут включать интерфейсы к базам данных, уведомления о событиях, средства лицензирования, службы безопасности и мониторы транзакций. Основная идея состоит в том, что приложение, которому потребовалось использовать службу некоторого объекта на этой же или удаленной машине, может это сделать с помощью посредника, без использования удаленного вызова процедур, требующего знаний о размещении серверного объекта. Брокер объектных запросов заботится о размещении и активизации зарегистрированных удаленных серверов, распределении запросов и ответов, обслуживании параллелизма и обнаружении дефектов связи. Доменные службы являются производственно-специфичными интерфейсами к общим бизнес-объектам. В рамках OMG предпринимались попытки определить подобные службы в таких предметных областях, как транспорт, здравоохранение, телекоммуникации, электронная коммерция, гуманитарные науки и финансы. Последняя сфера предполагает работу с банковскими счетами без наличия самого банка. Общие средства многократно используют или расширяют службы. Домены расширяют или многократно используют службы и общие средства. Приложения многократно используют и расширяют все эти группы и могут иметь собственные расширения.

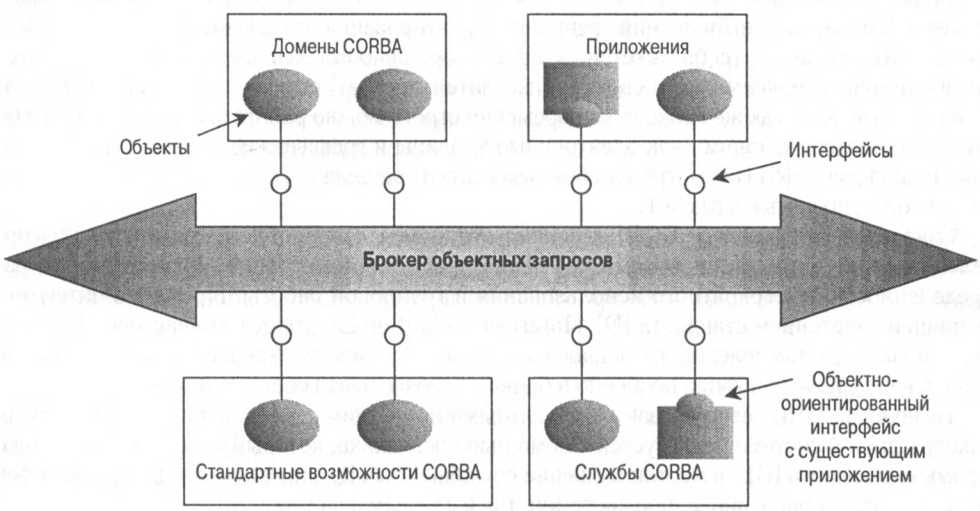


Рис. 4.4. Архитектура брокеров объектных запросов, предложенная группой OMG (Object Management Group)

Группа OMG определила также стандарт для интерфейсов — язык описания интерфейсов IDL (Interface Definition Language). С точки зрения семантики, IDL повторяет C++, но может использоваться только для определения интерфейсов. Это не полноценный язык программирования. В запросе к брокеру ORB указывается имя операции и параметры (которые

могут быть именами объектов). Брокер ORB организует обработку запроса, идентифицируя и запуская нужный метод и возвращая результат.

Брокеры объектных запросов могут обрабатывать запросы от всех типов объектов. Они связывают запросы с объектами и перенаправляют их другим брокерам. Брокеры ORB можно рассматривать как диспетчеры взаимодействия или как системные интеграторы, так как они либо маршрутизируют запрос к конкретному объекту, либо определяют синтаксис и семантику каждого запроса, поддерживая объектную модель. Они — маленький шаг в сторону интеллектуализации сетей. По мнению автора, в будущем экспертные системы и технологии компьютерного обучения должны сделать следующий шаг в направлении развития этой технологии.

Брокер ORB воплощает принцип абстракции данных на основе объектной модели. В этой модели **классы интерфейса** (interface class) определяют службы приложения, известные брокеру; а классы реализации представляют объектную оболочку. Класс интерфейса определяется в терминах его суперклассов, атрибутов и операций. Язык IDL поддерживает как динамическое, так и статическое связывание, в зависимости от требований к производительности и расширяемости. Все операции определения местоположения прозрачно обрабатываются брокером. Это делается с помощью классов реализации, однако технология CORBA не включает языка для их определения. Производители вольны осваивать эту область самостоятельно.

Технология ORB расширяет полиморфизм обычных объектно-ориентированных систем, основанный на наследовании, и позволяет пользователю во время выполнения программы выбирать нужный объект среди множества объектов, выполняющих идентичные функции. Таким образом, для выполнения функций редактирования пользователь может выбрать предпочтительный для себя текстовый процессор, основываясь на собственном опыте и предпочтениях и максимизируя свою производительность. В стандартизированной среде это было невозможно. Так же гибко и одновременно строго можно работать с другими общеизвестными средствами, такими как электронные таблицы и графические серверы. Таким образом, технология ORB (несмотря на свои недостатки) решила некоторые проблемы производительности, поднятые в главе 1.

Считается, что стандарт CORBA решает пять ключевых проблем, связанных с распределенными объектными системами, а именно проблемы интеграции, совместимости, распределенности, многократного использования и групповой работы. Проблема интеграции разрешена введением стандарта IDL (Interface Definition Language). До настоящего времени проблема распределенности решалась с помощью низкоуровневого программирования RPC, и каждый разработчик, особенно в банках, разрабатывал собственные каркасы.

Технология ORB тесно связана с объектными оболочками, так как доступ к объекту или пакету в другой системе реализуется с помощью посредника, который определяет и протокол в рамках стандарта IDL, и местоположение службы. Язык IDL позволяет задавать интерфейс как с объектно-ориентированной системой, так и с целым пакетом.

В версии 1 стандарта CORBA, вышедшей в 1991 году, язык IDL определен как протокол взаимодействия между объектами. В версии 2 (1994 года) определены способы взаимодействия брокеров объектных запросов от разных производителей. С точки зрения независимости от производителя, это — большое достижение. IIOP (Internet Inter-ORB Protocol) — это Internet-протокол взаимодействия между различными брокерами ORB в рамках стандарта CORBA. Это простой транспортный протокол, который позволяет гетерогенным ORB-продуктам взаимодействовать по протоколу TCP/IP. Проще говоря, он состоит из набора форматов сообщений. Общий протокол взаимодействия между брокерами ORB — GIOP (General Inter-ORB Protocol) от OMG — определяет способ отображения интерфейсов IDL

на сообщения. Протокол IIOP преобразует сообщения GIOP в TCP/IP, а значит они могут быть отправлены по локальной сети или через Internet. Адаптер портируемых объектов POA (Portable Object Adaptor) связывает ссылки на объекты с кодом необходимых служб, обеспечивая тем самым портирование.

Java содержит собственный, встроенный брокер объектных запросов под названием RMI (Remote Method Invocation). Он не совместим с CORBA, однако позволяет приложениям Java обращаться к службам других приложений Java как к локальным, независимо от их реального размещения. Для использования сервера, написанного на другом языке, программист должен создать на этом сервере Java-посредник, а после этого для связи с ним использовать интерфейс JNI (Java Native Interface). Для языка Java стандарт RMI выступает чем-то вроде собственного IDL. В настоящее время язык Java включает как протокол IIOP, так и совместимый с CORBA язык Java IDL.

Компания Microsoft предложила альтернативу технологии CORBA — свои модели компонентов COM (Common Object Model) и DCOM (Distributed Common Object Model). Они разработаны на основе механизма OLE (Object Linking and Embedding), который позволяет автоматически запускать приложения, если к созданному ими объекту производился доступ из другого документа. Технология COM предполагает непосредственную доступность локальных служб. Технология DCOM обеспечивает взаимодействие компонентов COM различных машин через механизм удаленного вызова процедур. Это решение менее элегантно и глобально, чем CORBA, однако его гораздо легче применить на практике. Это и сделало его более популярным среди программистов.

В CORBA 2 добавлена поддержка языков Ada, COBOL, C++, Java и Smalltalk. Кроме того, там имеются дополнительные функции для инициализации, реализации транзакций, безопасности, абстрактных интерфейсов, значений неопределенного типа, а также интерфейс DSI (Dynamic Skeleton Interface). Последний позволяет выбирать операции во время выполнения, а не жестко "зашивать" их в коде программы (см. ниже). Он дает возможность передавать объекты как по значению, так и по ссылке, а также обеспечивает интерфейс между COM и CORBA.

В 2000 году в версию CORBA 3 была добавлена поддержка брандмауэров и адресов URL, а также ряд функций, обычно ассоциировавшихся с MOM: асинхронная передача сообщений и управление очередями, а также функции реального времени. Версия CORBA 2 целиком рассчитана на синхронную модель, основанную на RPC. В версии 3 также была добавлена компонентная модель CORBA (CCM — CORBA Component Model), основанная на EJB (Enterprise Java Beans), однако независимая от языка. Была обеспечена поддержка взаимодействия с COM и OLE, а также взаимосвязь компонентной модели CORBA с управляющими элементами ActiveX и Java Beans.

Технология ORB позволила организациям легко внедрять архитектуры, основанные на службах. Хорошим примером является гибкая система ценообразования в одном из оптовых банков. В нем цены для различных дистрибьюторов формируются на основе собственных алгоритмов с учетом базовых цен. К сожалению, эта система обработки запросов на цены из различных пунктов продаж не дала ощутимых результатов. Это значит, что зря затрачивалась масса времени. Тогда банк создал систему, в которой трейдеры готовили отклонения и алгоритмы в виде электронных таблиц Excel и библиотек функций C++, вызываемых из этих таблиц. Результаты, предлагаемые на сервере, были доступны с помощью OLE другим пользователям Excel или лицам, использующим броузеры с поддержкой Java. Посредниками в этих операциях выступали брокеры ORB. В результате этот банк смог предложить массу служб, основанных на ORB, как пользователям, так и своим внутренним клиентам. Еще

одним примером может стать служба календаря, которая позволяет любому пользователю узнать режим работы бирж по всему миру. Архитектура, основанная на службах, устранила дублирование, неизбежное при реализации таких функций каждым из производителей в отдельности.

Роль программ среднего уровня в общей архитектуре программного обеспечения в наше время становится все более важной. Брокеры объектных запросов, мониторы транзакций и механизмы доставки сообщений играют решающую роль в создании устойчивых гибких систем. К сожалению, нам приходится работать в обществе, которое по-прежнему считает, что компоненты среднего уровня — это "программы, за которые никто не хочет платить".

Одним из первых коммерческих продуктов на основе технологии ORB была неудачная система DOMS от компании Hyperdesk. На самом деле создатели этого продукта следовали рекомендациям группы OMG. Они попытались создать замену устаревшим продуктам автоматизации офиса от компании Data General (DG). К сожалению, в качестве основы этой работы была неудачно выбрана технология NewWave от HP. В конце 1980-х эта группа выделилась из DG в отдельную компанию, а Крис Стоун (Chris Stone) вернулся к формированию OMG. В это время такие компании, как Sun, HP, DEC и NCR, уже имели собственные идеи решения проблемы вычислений в распределенных офисах.

Существовало два глобальных подхода: статический от HP и Sun и (менее эффективный) динамический от Hyperdesk и DEC. Динамическая модель требовала создания единого интерфейса API для построения запросов и единого механизма распаковки сообщений. В статической модели на каждую операцию отводился отдельный фрагмент программы, и все запросы выполнялись различными подпрограммами. Каждый объект встраивался в каркас, связанный с отдельной операцией. Каждый каркас передавал параметры, как будто источником запроса была подпрограмма. Технология CORBA обеспечила интеграцию статической и динамической моделей посредников объектных запросов, поэтому любое CORBA-совместимое решение должно поддерживать оба подхода.

Самым популярным брокером объектных запросов является Orbix от Iona Technologies. Другие известные продукты — CORBA Plus от Ventel, Component Broker от IBM, Visibroker от Inprise, DAIS от Peer Logic и Weblogic Enterprise от BEA.

Некоторые продукты ORB предлагают интересные функции регистрации, позволяющие поддерживать широковещательные сообщения без методологии "классной доски". Целевому объекту отправляется сообщение, регистрирующее интерес какого-то другого объекта. Если у целевого объекта происходит соответствующее событие, он отправляет сообщение заинтересованному объекту.

Некоторые продукты используют таблицы службы каталогов, структурированные подобно стандарту электронной почты X500. Ссылки на распределенные объекты получаются из каталогов как посредники (суррогатные объекты), которые затем копируются в пространство задачи источника запроса. Сообщения передаются посредникам, что обеспечивает истинную прозрачность объектов. На самом деле за это отвечает диспетчер распределенных объектов. Модель вызова удаленных методов показана на рис. 4.5.

Язык IDL используется для определения атрибутов и операций интерфейсов (их подписей). Они объединяются в опорные объекты (stub), каркасы и файлы определений. Опорный объект является клиентской частью распределенного объекта или объектом-посредником, который получает запросы локально (посредством обычного локального вызова функции), сортирует параметры и отправляет сообщение реальному объекту. После этого каркас получает сообщение, сортирует аргументы и вызывает целевой метод напрямую. Каркасы должны быть заполнены кодом приложения.

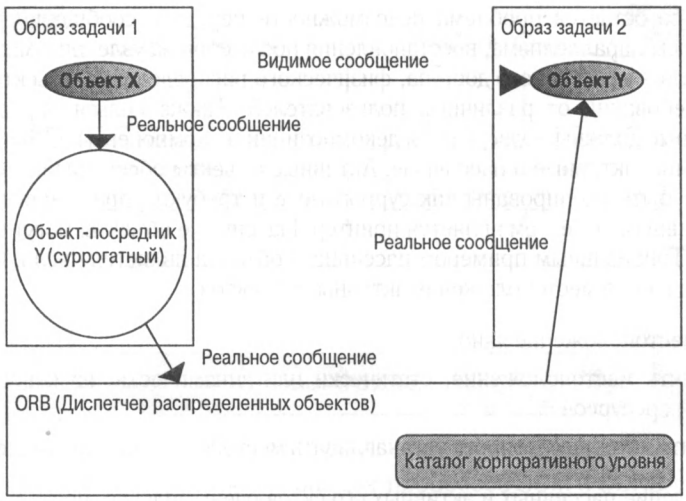


Рис. 4.5. Вызов удаленного метода

Термин **сортировка** (marshalling) оставался для меня загадкой до тех пор, пока я не узнал, что американцы называют станции формирования составов сортировочными (marshalling yard). И тогда сформировался образ локомотива, который сортирует вагоны по порядку и формирует готовый к отправке на другую станцию состав. Аналогично, когда брокер ORB находит подходящий сервер, он сортирует параметры и представляет их в формате, пригодном к отправке. Естественно, исключения и сообщения об ошибках возвращаются опорному объекту для передачи их клиенту. На рис. 4.6 показаны взаимосвязи между различными компонентами CORBA.

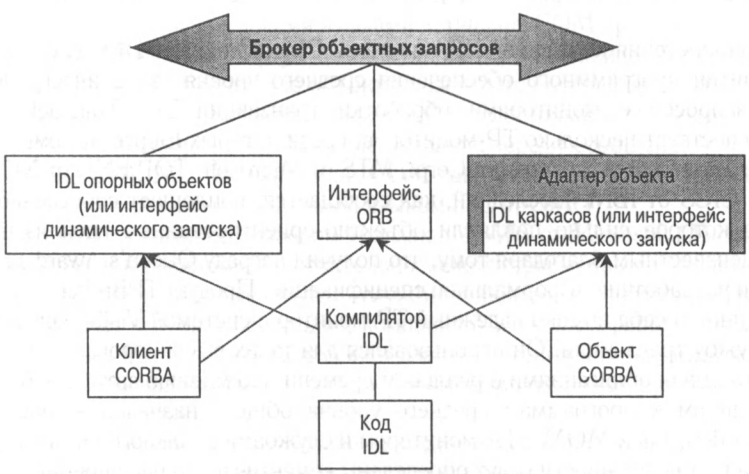


Рис. 4.6. Взаимосвязь CORBA и IDL

Тем не менее остается проблема невозможности передачи сообщений напрямую. Она включает вопросы параллелизма, восстановления после сбоя на узле, оптимизации размещения объектов для эффективного доступа, физического распределения объектов, разрешения конфликтов требований от различных пользователей. Также нельзя определить размеры объектов, которые должны подвергаться декомпозиции на компоненты. Объекты можно разделить на два типа: активные и пассивные. Активные объекты обеспечивают услугами другие объекты, могут быть скопированы как суррогатные и требуют управления параллелизмом. Например, активным объектом является принтер. Пассивные объекты могут быть физически распределены. Тривиальным примером пассивного объекта является число. Существует три стратегии определения местоположения активных объектов.

1. Задать местоположение явно.
2. Определить местоположение, статически или динамически, на основе требуемых и доступных ресурсов.
3. Разрешить источнику запроса устанавливать местоположение динамически.

Такое разделение пассивных и активных объектов очень полезно, поскольку это позволяет определить требования к обработке активных объектов и отобразить их на физическое оборудование, после чего может приниматься решение относительно отображения пассивных объектов.

Соответствие таким стандартам, как CORBA и CORBA Services от OMG, обеспечивает переносимость приложений на различные платформы. Все преимущества объектной технологии (многократное использование, расширяемость, локализация поддержки, комплектация приложений из программных интегрируемых контуров (software integrated circuit — IC) и т.п.) были реализованы в распределенной среде. Этот подход обеспечивает масштабируемость систем, размещенных на нескольких компьютерах на базе различных распределенных архитектур. Сложные сетевые интерфейсы и API были заменены мощными прозрачными механизмами управления сообщениями, размещением объектов и глобальной политикой их именования.

После приобретения компанией BEA системы ObjectBroker от DEC следующим направлением развития программного обеспечения среднего уровня стала интеграция брокеров объектных запросов с мониторами обработки транзакций TP (Transaction Processing Monitor). Существует несколько TP-мониторов, среди которых Ellipse от компании Cooperative Solutions Inc., Encina от Transarc Corp., MTS от Microsoft, TOP END от NCR, Tuxedo от BEA и даже CICS от IBM. Последний, как сообщается, воплощает собственное проектное решение, на которое сильно повлияли объектно-ориентированные идеи языка Smalltalk. Он стал общеизвестным благодаря тому, что получил награду Queen's Award за использование командой разработчиков формальной спецификации. Продукт TPBroker от фирмы Hitachi также объединил в себе высоко надежный TP-монитор с системой Visibroker, имеющей собственную службу транзакций. Он использовался для таких высокопроизводительных приложений, как торговля облигациями в реальном времени. По мнению автора, в будущем рынок повернется лицом к программам среднего уровня общего назначения, обеспечивающим функции как ORB, так и MOM с TP-монитором и службами реального времени. Для некоторых отраслей промышленности будут определены семантические расширения, возможно, основанные на XML, которые также будут стандартизированы.

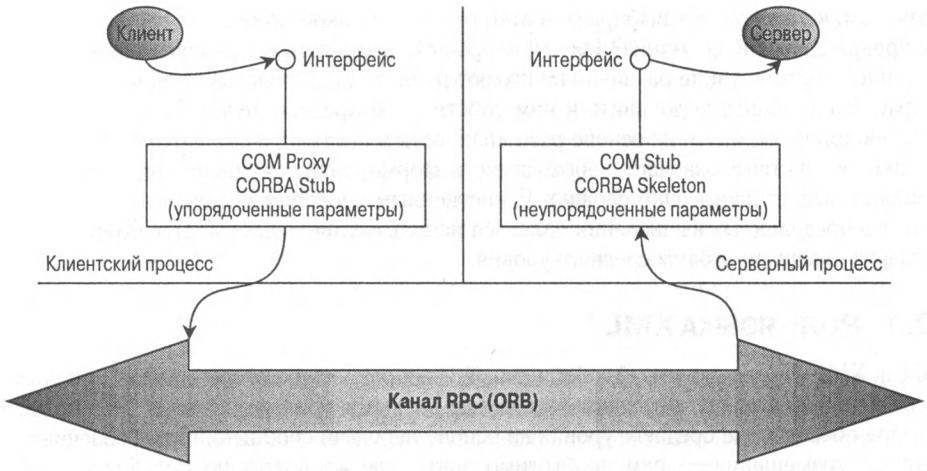


Рис. 4.7. Сортировка в рамках процессов COM и CORBA

Брокеры ORB являются основой для построения архитектур на базе служб с использованием компонентов в виде "черных" ящиков, так как в данном случае единое адресное пространство отсутствует.

COM-технология

Основной альтернативный по отношению к технологии CORBA подход был применен компанией Microsoft для реализации компонентов COM+ на основе технологий COM, DCOM, MTS и ActiveX. Технология COM определяет двоичный стандарт для интерфейсов и поэтому является независимой от языка. Технология CORBA не предполагает какого-либо конкретного размещения компонентов в памяти и достигает языковой независимости за счет определения взаимодействия со многими языками. Стандарт COM предполагает встраивание интерфейсов подобно таблицам виртуальных функций C++. Эквивалент ORB называется каналом RPC (RPC Channel). Опорные объекты называются посредниками (proxy), а каркас CORBA соответствует опорным объектам (stub) COM. В остальном эти подходы аналогичны (рис. 4.7). Например, технология COM поддерживает очереди сообщений и уведомления о событиях с помощью MSMQ. В технологии COM также определен собственный язык IDL (его не следует путать с абсолютно отличным языком IDL от OMG).

Технология COM+ связана с сервером транзакций MTS от Microsoft (Microsoft Transaction Server), поэтому и не является действительно межплатформенным стандартом (частично из-за того, что эта и некоторые другие службы никогда не поддерживались мостами COM/CORBA). Чтобы исправить ситуацию, компания Microsoft разработала протокол SOAP (Simple Object Access Protocol) для поддержки взаимодействия компонентов через Internet. Это позволило разработчикам строить трехуровневые компонентные архитектуры на любой платформе и интегрировать их.

Распределенные вычисления привнесли дополнительные проблемы, не свойственные централизованному подходу. Например, каждый пользователь в отдельно взятом офисе может работать с собственным определением такого понятия, как заказчик. В общем случае заказчиком может быть любая сторонняя организация или сотрудник компании. Кто является

заказчиком, когда ребенок выбирает, а отец платит? В какой момент потенциальный покупатель превращается в заказчика? На эти вопросы сложно дать однозначный ответ. В централизованной системе такие различия не играют роли, так как пользователь может взять нужные ему копии данных и добавить к ним собственное представление. Распределенная сущность никогда не строится на основе разобщенности и противоречий. Поэтому переход к распределенной системе означает необходимость формирования общепринятых определений, что иногда бывает достаточно сложно. Единственным способом преодолеть описанные проблемы распределенных вычислений является использование языка XML (eXtensible Markup Language) поверх программ среднего уровня.

4.2.1. Роль языка XML

Язык XML — расширяемый, обобщенный язык разметки, который может использоваться для описания данных, передаваемых между разными приложениями и системами. Программное обеспечение среднего уровня на основе передачи сообщений не обеспечивает прозрачности размещения — вам необходимо знать, где расположено целевое приложение. Таким образом, чтобы продвинуться дальше обычного соединения "точка-точка", необходим маршрутизатор. Язык XML создан на основе более старого языка SGML (Standard Generalized Markup Language) и сходен с HTML, за исключением того, что данные могут описывать сами себя с использованием иерархии дескрипторов. Это приближает стиль Web-страниц к программам. XML позволяет отправителям сообщений отделять заголовок от содержимого. Поэтому вместо явного указания на выделение некоторого важного фрагмента текста жирным шрифтом мы просто заключаем его в дескрипторы `<emphasis>` и используем таблицу стилей XML для определения значения `<emphasis>` (например, выделение текста жирным шрифтом или другим стилем форматирования). Например, в следующем фрагменте дескрипторы определяют структуру сообщения (от кого, кому, тело сообщения), его форматирование (курсив) и часть его содержания (заказчик).

```
<?xml version= "1.0"?>
<!DOCTYPE "sales memo">
<from>Susie Eizus</from>
<to>Eric Cire</to>
<body>Мы только что получили срочный заказ от
<customer>Imperfect Palidromes Inc.</customer>.
Пожалуйста, выполните его сегодня.
...
</body>
```

Анализаторы XML позволяют определить, какие типы дескрипторов могут быть в документе (их допустимость) по описанию типов документа DTD (Document Type Descriptor). Тот факт, что некоторые поля документа могут быть необязательными, означает, что получатель должен иметь возможность извлечь данные из сообщения XML с пропущенными полями. Поэтому если существующая база данных содержит одно определение заказчика, а новая система — другое, они все равно могут взаимодействовать. В результате одним из основных применений XML является информационный обмен между системами с различными форматами хранения одной и той же информации, возможно, даже на разных платформах. Его можно использовать поверх программ среднего уровня для создания корпоративных стандартов обмена данными и для оболочек существующих приложений. Такой тип приложений получил

название EAI (Enterprise Application Integration). XML также привнес потенциал электронного обмена данными EDI (Electronic Data Interchange) в сферу мелких поставщиков. В таких приложениях некоторые дескрипторы рассматриваются как метаданные.

Существует ряд стандартов метаданных, например Meta Content Framework от Netscape, предназначенный для описания содержимого Web-узлов. В этом контексте можно представить себе дескрипторы `<Geology>`, `<Shakespeare>` и даже `<useful for wannabe bomb makers>`. Язык XML очень гибок и может использоваться при определении новых, специализированных языков для различных предметных областей. Например, он применялся для определения редактора формул, а также для языков электронной коммерции. Он может поддерживать множество языков, так как использует 16-битовый набор символов Unicode, а не изначально 7-битовый набор кодов ASCII, позволяющий описать 128 символов.[2] Таблицы стилей XML обычно пишутся на другом расширяемом языке — XSL. Существует также язык определения пользовательского интерфейса XUL, использующий древовидную структуру для определения элементов UI, что позволяет размещать их в броузерах "на лету".

Самоописательная, иерархическая структура дескрипторов позволяет реализовать более богатые средства доступа к данным, чем существующие в обычных базах данных, например контекстно-зависимые запросы и элементы навигации. Система управления содержимым POET представляет собой пример приложения, построенного на XML. Приложение подобного вида, с которым я работал ранее, обеспечивает перевод ярлыков и описаний продуктов с многочисленных иностранных языков. Язык XML позволяет трактовать документы как объединения небольших разделов — вплоть до стандартных фраз. В сочетании с автоматическим управлением версиями такой подход значительно сократил количество создаваемых документов и их объем. Программа Astoria от Xerox — еще один продукт, разработанный специально для такого типа приложений. Существует еще и группа XML-серверов, в том числе eXleon от ODI и Tamino от Software AG. Эти продукты расширяют возможности объектно-ориентированных баз данных, включая в них объектную модель, основанную на XML. В настоящее время на рынке предлагается множество анализаторов и редакторов XML, часть из которых бесплатна.

Можно сказать, что XML даже слишком гибок. Необходимо удостовериться, что сообщение отражает конструкции используемого языка и одновременно остается максимально простым. Один из подходов к решению этой задачи состоит в создании внутреннего стандартного подмножества XML с определенными дескрипторами, в которые можно конвертировать все сообщения от существующих систем. Такой подход был реализован в банке Chase Manhatten Bank [606]. Заказчикам потребовалось связать приложения, работающие на разных платформах и использующие множество языков, от COBOL и RPG до C++ и Java. Каждая из систем имела свой формат сообщений фиксированной длины. Действующие до этого средства передачи данных по FTP были дорогостоящими и часто приводили к ошибкам. Банку требовалась гарантированная доставка, и поэтому был использован MOM-продукт MQ Series от IBM, определяющий свой формат сообщений с помощью XML. Это позволило банку определить стандартные форматы сообщений для таких операций, как движение средств, со многими необязательными полями, перекрывающими все существующие форматы. Хотя в этом банке широко использовалась технология CORBA, она не могла обеспечить передачу данных в формате, принятом в MQ Series или ASN.1. Ни один из предлагаемых форматов не был читабельным для человека. Банк Chase определил некоторые ограничения для встроенного анализатора. Естественно, был введен ряд правил, касающихся "сохранности

---

[2] Несложно убедиться, что n-битовый код позволяет представить 2^n символов.

типов" (например, ни один элемент сообщения не может быть определен как ANY или EMPTY, и все элементы должны иметь свой тип). Это типичное приложение, построенное на XML, и аналогичные проекты в других банках приводят к созданию аналогичных стандартов языков. Open Financial Exchange (www.OFX.net) — это набор стандартных типов финансовых элементов, представляющих такие данные, как номера кредитных карточек или типы транзакций. Еще один стандарт — Open Trading Protocol (www.OTP.org) — касается электронной торговли.

Еще одно впечатляющее приложение, построенное на XML, позволяет виртуальным читателям новостей улыбаться, хмуриться или злиться, в то время как синтезатор читает текст. Речевые выражения закодированы в дескрипторах текста. Соответственно, изменяется и тон голоса искусственного диктора. Этот продукт расширяет идею *аватаров* (avatar), довольно известную в чатах и изначально описанную еще в 1992 году в прекрасном научно-фантастическом романе [727].

Сервер BizTalk от Microsoft — это каркас, основанный на формате XML. Множество компаний используют его в качестве основы многих приложений. Сервер BizTalk включает схемы таких бизнес-процессов, как обслуживание каталогов продукции, предложения и закупки. Он маршрутизирует сообщения и может содержать правила определения порядка работ. XMI (XML Metadata Interchange) — это стандарт OMG, который призван объединить UML, MOF и XML. Он использует описания типов документа XML для определения компонентов MOF и моделей UML. MOF-совместимые модели являются взаимозаменяемыми в XMI-совместимых хранилищах.

Можно подумать, что XML специально создавался как универсальный язык выразительных сообщений, передаваемых между разрозненными системами и даже целыми корпорациями. Правда, само наличие XML не означает, что больше корпоративных программ начнут понимать друг друга.

4.3. Интеграция приложений предприятия

В 1990-х годах многие предприятия предприняли важные изменения в технологии и принципиально перешли от централизованных систем, основанных на больших машинах (обычно от одного поставщика), к распределенным. Как для разработчиков, так и для приложений более удобными оказались платформы рабочих станций. Это связано с их возрастающей мощностью и простотой графических интерфейсов. Этот процесс можно ассоциировать с ускорением реакции на запросы потребителей со стороны разработчиков. Однако существование непереносимых старых систем означает, что в переходный период ключевым вопросом является совместимость. Разработки аппаратных средств и реляционных баз данных также испытывают нарастающую потребность в распределенной архитектуре, обеспечивающей эту совместимость. Многим предприятиям приходится работать с мини-ЭВМ, обособленными или объединенными в сеть персональных компьютеров с текстовыми процессорами, электронными таблицами и, наконец, броузерами. На больших машинах работали как реляционные, так и нереляционные базы данных, а также приложения, основанные на работе с записями и написанные на языках COBOL или Assembler.

По мере появления новых приложений, во главу угла стал вопрос доступа к информации, хранящейся на больших машинах. Операции обращения к данным можно было выполнять по ночам, однако лучшим решением стало построение моста на основе программного шлюза,

конвертирующего данные между системами, например между системами IMS на большой машине и Oracle или Sybase на UNIX-серверах. Часто такое программное обеспечение поставляют сами производители реляционных баз данных. Одним из первых таких продуктов был NetGateway от Sybase. В результате можно извлекать и изменять данные, не прибегая к обычному копированию. Это избавляет от необходимости перестраивать терминальные экраны ввода данных для новых приложений. Такой подход обеспечил для пользователей достаточную прозрачность размещения и сделал взаимодействие с базами данных значительно проще.

Среди сложных вопросов перехода на новые технологии, с которыми столкнулось большинство организаций, были проблемы сетевой архитектуры, надежности и управления. Одна организация, с которой я работал, рассматривала свои сети как иерархию уровней служб. На самом верхнем уровне пользователи могли подключать свои рабочие станции к различным локальным серверам. Следующий уровень реализовывал поддержку сервера рабочей группы. Оба уровня основывались, в основном, на технологии локальных сетей, а на уровне рабочей группы использовалась матричная коммутация. Третий уровень соединял первые два с серверами отделов и баз данных, используя концентраторы Ethernet. И наконец, существовал довольно массивный и сложный уровень связывания, поддерживающий шлюзы к существующим системам на больших машинах, доступ к глобальной сети и маршрутизацию сообщений. Серьезным вопросом для служб финансового сектора является безопасность, и эта компания подошла к данному вопросу достаточно серьезно. В своей сетевой системе управления она реализовала стандарты служб безопасности DCE от Open Software Foundation и протокол аутентификации Kerberos. Для соединения локальных сетей различного типа (например, Ethernet и Token Ring) и поддержки брандмауэров между сегментами сети были задействованы маршрутизаторы. Эта распределенная архитектура обеспечила создание ряда сложных, важных для производства и гибких приложений в течение двух лет. Стандартные системы обеспечили перемещение финансовой информации из базы данных большой машины и сочетание этой информации с данными подсети маркетинга. Затем эти данные помещались в серверные реляционные системы, проверялись в интерактивном режиме, возвращались на большую машину и одновременно передавались на локальные рабочие станции. Другие системы обеспечивали целостные прямые связи между ранее несовместимыми офисными системами.

Сегодня термин "интеграция приложений предприятия" (EAI — Enterprise Application Integration) используется для обозначения интеграции разрозненных систем, пакетов и компонентов, задействованных в ключевых производственных процессах, т.е. для описания различных цепочек программ, совместно работающих в рамках одного предприятия (в идеале они изначально должны быть разработаны как компоненты). Таким образом, проекты EAI требуют многих из описанных выше средств: программ среднего уровня, метаданных, форматов сообщений и т.д.

Любая организация и ее программное обеспечение развиваются на протяжении многих лет, в течение которых определяется множество точек соединения между программными пакетами различного назначения. Система в целом (как и само производство) является негибкой, так как все ее связи не могут быть реорганизованы без больших усилий; а определения точек соединения специфичны для связываемых модулей. Такая архитектура не допускает никаких обобщений. Аналогичная ситуация возникает при использовании различных систем в разных подразделениях или дочерних компаниях. Переход к электронной коммерции потребовал более тесной интеграции. В качестве решения этой проблемы можно определить набор общих соединителей — язык, на котором могут общаться все компоненты. Это облегчит их

реорганизацию. Это не просто вопрос выбора технологии, такой как MOM, CORBA или XML. Помимо технологии необходимо выбрать общую бизнес-модель и соответствующие протоколы. Речь идет не только об интеграции данных разрозненных систем или создании хранилищ данных. Должны также интегрироваться и функции самих систем.

Давление конкуренции, новые методы работы и переход к электронной коммерции показали, что необходима интеграция данных и процессов в реальном времени. Ночной переброски данных и репликации при этом недостаточно. За пределы одной организации могут выходить не только процессы — вся работа может быть распределена между несколькими компаниями. Все это требует немедленной реализации. Необходимо преобразовать не только основные форматы данных, но и понятия самого производства. В этой связи часто упоминается задача преобразования 2-значной даты в 4-значную, однако унифицированное отображение в системе информации о клиентах, имеющих различное местоположение, — задача гораздо более сложная. Пакеты программ зачастую были довольно негибкими и несовершенными, чтобы обеспечить решение этой задачи. Интеграция приложений предприятия требует внимания к созданию общих производственных стандартов и унифицированной технологической архитектуры. Также нужны ясные раздельные интерфейсы к модулям. Производители пакетов обещали все это создать, но, естественно, только в том случае, если вы закупите большую часть необходимого программного обеспечения именно у них. Такой путь мог привести в будущем к еще большей проблеме интеграции.

На практике можно было бы остановиться на технологии интерфейса ядра: стандартизировать один из COM-интерфейсов от Microsoft, EJB или CORBA. Но даже в этом случае должно быть достигнуто полное соглашение не только по технологической архитектуре. На рис. 4.8 показаны различные технологии и понятия, применяемые по отношению к архитектуре.

Рис. 4.8. Уровни архитектуры

Программные продукты EAI должны поддерживать уведомления о событиях, управление потоками, маршрутизацию сообщений на основе правил, а также трансформацию данных. Последняя предполагает создание средств для определения интерфейсов и преобразования форматов данных, а также создание хранилища всего этого. На рынке систем EAI представлены

пакеты, программы среднего уровня и оболочки, средства автоматизации производства, шлюзы к базам данных и серверы приложений. Серверы приложений варьируются от интегрированных решений для электронной коммерции, таких как Websphere от IBM или Weblogic от BEA, до систем кэширования баз данных. Также известны средства разработки приложений, такие как UNIFACE от Compuware и Forte Fusion от Sun. Продукт Fusion базируется на кросс-платформенной компонентной системе интеграции и управления производством Forte Conducto, а также объектно-ориентированных языках четвертого поколения TOOL и SynerJ. Для обмена сообщениями эта система использует XML, а для представления и обработки правил — XLS. Системы связаны через приложения-посредники, которые общаются посредством соединителей (connector) с программными интерфейсами на языках C, TOOL или XML.

Введение связки для объединения существующих приложений может, конечно, сказаться на производительности и существенно усложнить тестирование и выявление ошибок.

Продукты EAI не обеспечивают готовых решений. Необходимо разработать или адаптировать интерфейсы, процессы и новые компоненты, а это может стать тяжелой программистской задачей. Со многих точек зрения технология EAI, в отличие от технологии компонентной разработки CBD, предлагает всего лишь другой взгляд на тот же набор объектов. Компонентная разработка предполагает создание составляющих элементов, которые можно легко объединить. На практике большинство разработок представляет собой смесь этих двух подходов. Более детально технология компонентной разработки CBD будет рассмотрена в главе 7.

Ключевыми техническими свойствами среды EAI являются распределенность, связность и гибкость. Эти факторы увели развитие компьютерной техники прочь от напоминающих водонапорные башни больших машин, и люди вдруг осознали, что для осуществления революционных преобразований и описания мира распределенности, параллелизма и сообщений важна не только расширенная среда объектно-ориентированного программирования. При этом нужно привлечь объектно-ориентированные подходы к анализу и проектированию. Без хорошего объектно-ориентированного анализа и культуры управления проектами распределенное решение может стать еще более громоздким для управления, чем замененные им большие машины. Определение такого подхода стало ключевой задачей компаний, вовлеченных в процесс EAI.

4.4. Стратегии перехода к объектной технологии

Многие убедились в целесообразности смещения направлений разработки систем в сторону объектно-ориентированного компонентного подхода. Возможно, это произошло потому, что другие компании пошли по этому пути и появилась мода на объектные технологии (OT) и компонентную разработку CBD. Позднее, сбившись с пути и неудачно завершив проект, часть этих компаний просто смогла использовать накопленный опыт и применить знания существующих систем и приемов программирования для создания объектных моделей. Для замены существующих старых систем есть несколько причин. Например, производитель пакетов может заметить тенденцию перехода к объектной технологии и модифицировать свой продукт, пытаясь опередить своих конкурентов на рынке. Он может повысить конкурентоспособность своей системы, оснастив существующий продукт графическим интерфейсом, информационной системой управления (MIS — Management Information System) или способностью работы в распределенной среде. Пользователи могут захотеть получить

преимущества новых стандартов, более дружественных интерфейсов, функций электронной коммерции наряду с преимуществами самого объектного подхода, которые были рассмотрены в главе 2. В последнем случае существенно то, что системы электронной коммерции (для торговли с другими фирмами, для работы с заказчиками или для интеграции цепочки поставки) масштабируемы по определению, так как количество их пользователей непредсказуемо. Возможно, свою роль сыграла и большая гибкость при условии все более быстрого изменения требований. Здесь имеется в виду как гибкость самой объектной технологии, так и требование совместимости с изменениями объектов и производственного процесса. Новые компании на рынке электронной коммерции (или их дочерние предприятия) не сталкиваются с проблемой интеграции существующих систем, а вынуждены строить всю конструкцию с нуля. С другой стороны, они не сталкиваются с прогнозируемыми проблемами сопротивления изменениям, которые приходится решать более опытным компаниям.

Как производители, так и пользователи заинтересованы в снижении стоимости проектов на разработку и адаптацию программных продуктов, а также в сокращении сроков выхода на рынок. Мы уже видели, что ОТ при ее грамотном применении может послужить обеим целям. Однако моментальный переход на эту технологию очень желателен, но вряд ли возможен. В то же время постепенный переход может продолжаться так долго, что его преимущества потеряют всякий смысл. Типичным решением является многократное использование существующих компонентов или даже целых систем и пакетов. Кроме того, отдельные производители бухгалтерских программ и систем управления производством на базе технологии ERP (Enterprise Resource Planning) пытаются разделить свои продукты на компоненты, работоспособные в среде других продуктов. Принципы такой компонентной разработки будут отдельно рассмотрены в главе 7. Итак, существует несколько возможностей: совмещение, повторное использование, расширение, а также мгновенный или постепенный переход на новую объектную технологию. Эти варианты тесно взаимосвязаны, поэтому вначале рассмотрим совмещение.

В этом разделе мы ознакомимся с различными стратегиями, которые удовлетворяют требованиям организаций по решению проблем перехода и совместимости. Они будут рассмотрены с точки зрения разработчика, который стремится создать объектно-ориентированное приложение, требующее использования служб, реализованных на основе других стилей программирования, таких как экспертные системы, языки четвертого поколения (4GL), библиотеки процедур (в том числе NAG FORTRAN), системы параллельной обработки, реляционные базы данных и даже нечеткие контроллеры. Какие фундаментальные вопросы возникают при использовании, проектировании и построении средств взаимодействия? Как работать с приложениями, написанными на языках COBOL и Assembler? Какова роль объектно-ориентированного анализа и компонентного проектирования в таком переходе? Существует ли стратегия, позволяющая превратить существующее процедурное приложение в объектно-ориентированное без нарушения служб, предоставляемых существующим пользователям? Какова роль средств EAI?

В этом разделе мы рассмотрим такую проблему, как создание оболочки для старого приложения, существующего в различных версиях. Мы также рассмотрим способы использования объектно-ориентированного анализа в частности и объектной технологии в целом в качестве метода такого перехода.

4.4.1. Совмещение объектно-ориентированных систем с обычными

Существует масса сценариев, с помощью которых объектно-ориентированные приложения можно совместить с существующими обычными системами. Среди них нужно отметить следующие.

- Последовательный переход от существующих систем к будущим объектно-ориентированным реализациям, в которых части старых систем будут временно продолжать использоваться.
- Развитие и интеграция существующих корпоративных систем, которые слишком сложны или громоздки, чтобы переписать их сразу. В таких структурах части старых систем могут существовать неопределенно долгое время (EAI).
- Необходимость построения решений на существующих пакетах.
- Многократное использование высокоспециализированных или оптимизированных процедур, встроенных экспертных систем и программного обеспечения, специализированного для работы с конкретными аппаратными средствами.
- Совместное использование лучших из существующих реляционных баз данных в одной части приложения с применением языков объектно-ориентированного программирования и(или) объектно-ориентированных баз данных в его другой части.
- Построение графических оболочек или броузеров для существующих систем.
- Совместная обработка и архитектура "классной доски" позволяют привлекать для работы с новыми объектами уже существующих агентов.
- Потребность в согласовании действий с существующими системами в пределах локальной или глобальной сети.

Главный вопрос состоит в том, как правильно организовать переход от огромных систем, которые чрезвычайно дороги и сложны в настройке. Первой рекомендуемой здесь стратегией является построение так называемой объектной оболочки (object wrapper). Объектная оболочка может использоваться для перехода к объектно-ориентированному программированию, одновременно с этим позволяя сохранить инвестиции, вложенные в приложения обычного типа. Понятие оболочки в объектно-ориентированном мире стало частью фольклора, и, насколько известно автору, первое упоминание этого термина связывают с именем Уолли Дитриха (Wally Dietrich) из компании IBM. Иногда это первенство отдают, хотя и несколько в ином контексте, разработчикам языка Objective-C Брэду Коксу (Brad Cox) и Тому Лаву (Tom Love). Утверждают также, что этот термин был в моде в компании IBM еще в 1987 году.

Сложно оспаривать большие инвестиции в коммерческие пакеты и программы COTS (Commercial, Off-The-Shelf), написанные на обычных языках, таких как COBOL, Assembler, PL/1, FORTRAN и даже APL. Хотелось бы отметить, что большая часть средств, вложенных в эти перешедшие по наследству программы, была инвестирована именно в их поддержку. Это связано с тем, что в традиционных системах любое изменение структуры данных требует проверки реакции на это изменение практически каждой функции. Как было показано в главе 1, для объектно-ориентированных систем это не характерно благодаря инкапсуляции

структур данных в рамках функций, которые их используют. Однако, как бы мы ни хотели полностью заменить эти системы, экономические расчеты не позволят этого сделать, если речь идет о больших объемах кода. У нас в распоряжении просто не окажется таких громадных ресурсов разработчиков. Поэтому приходится основываться на уже вложенных инвестициях и постепенно переходить в прекрасный мир объектной технологии и компонентов.

Вполне реально создать оболочку для громадного пакета, а затем постепенно менять или отбрасывать его составные части. Построение оболочек помогает защитить инвестиции в старые системы в процессе перехода к объектно-ориентированному программированию. Объектные оболочки обеспечивают взаимодействие новой, объектно-ориентированной части системы с традиционными программами посредством передачи сообщений. Сама оболочка может быть написана на том же языке программирования, что и программа, например на COBOL, либо для нее можно использовать программы среднего уровня, о которых шла речь в предыдущем разделе. Это может потребовать частичных инвестиций, и как только они будут сделаны, практически никакая поддержка не понадобится, по крайней мере — теоретически.

Представим себе, что существующая система на языке COBOL взаимодействует с пользователем посредством традиционной системы меню. На каждой странице предлагается порядка 10 пунктов меню, а в дочерних узлах дерева меню содержатся обычные управляющие элементы, используемые при вводе данных. Такой порядок характерен для большинства систем прошлых лет. Оболочка должна обеспечить выполнение всех функций старой системы через объектный интерфейс. Это показано на градиграмме[3] на рис. 4.9, где маленькие прямоугольники на границе оболочки представляют ее видимые операции, которые на самом деле связаны с функциями старой программы, и таким образом получают доступ к ее данным. В сущности, оболочка — это большой объект, методы которого "реализуют" функции меню старой системы. Единственное отличие между новым объектом и старой системой состоит в том, что объект может отвечать на сообщения от других объектов. А это уже обеспечивает некоторые преимущества, которые проявляются при обнаружении ошибки либо получении запроса на изменение или добавление новой функции. Так как в старую систему вмешиваться нельзя, приходится создавать набор объектов, реализующих новую функцию. Старые пользователи не заметят отличий при использовании традиционных функций, хотя их вызовы будут передаваться через оболочку. Оболочка может быть большой или маленькой, однако в контексте совмещения оболочки имеют тенденцию разрастаться. Для систем, управляемых из командной строки, оболочка может представлять собой набор командных файлов или сценариев операционной системы. В ее роли может также выступать набор интерфейсов CORBA или COM+ IDL. Если в старой системе использовались формы или экранный интерфейс, оболочка может содержать программы, которые записывают данные на экран и считывают их. Для этого можно использовать виртуальный терминал. Такой подход гораздо проще реализовать на машинах VAX. Это не всегда возможно в таких операционных системах, как OS/400, где может потребоваться специализированное программное обеспечение, брокеры объектных запросов или протокол Java RMI. Новые функции или модификации старых можно реализовать через новые объекты со своими собственными инкапсулированными структурами данных и методами. Если потребуются службы старой системы, их можно запросить,

---

[3] Термин "градиграмма" был придуман для обозначения диаграмм, в которых операции представлены маленькими блоками на границе прямоугольника, представляющего используемый объект. Такой тип диаграмм был введен в 1980-х годах Гради Бучем (Grady Buch) в его работе, посвященной языку Ada. Сегодня градиграммы используются в UML для представления диаграмм модулей.

передав сообщение. После обработки запроса результат работы старой программы дешифруется оболочкой и передается источнику запроса.

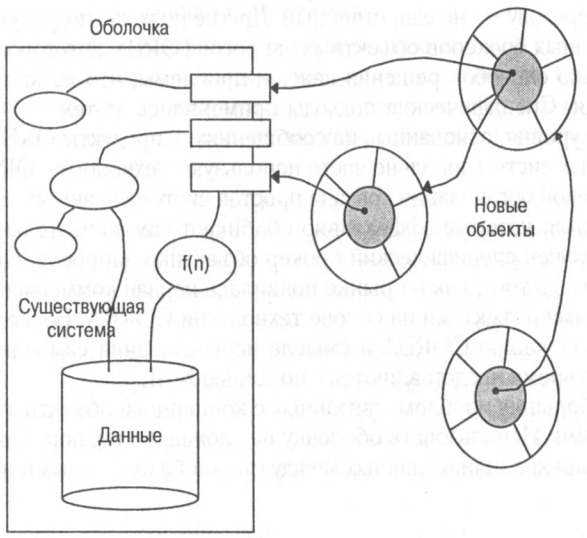


Рис. 4.9. Объектные оболочки

Однако хватит рекламы! Реализация оболочек не так проста в некоторых отношениях, как это кажется на первый взгляд. Большая часть литературы, посвященной оболочкам, пытается донести до нас необходимость применения брокеров объектных запросов. Когда по некоторой причине они не доступны, программисты сталкиваются с непосредственной реализацией деталей. Одна из проблем при этом касается масштаба объектов (granularity). Большинство теоретических аргументов и добрая часть практического опыта программистов на объектно-ориентированных языках свидетельствуют о том, что маленькие объекты чаще соответствуют конкретным требованиям, чем большие. Вспомним рекомендации по объектному проектированию из главы 2: интерфейс должен быть настолько узким, насколько это возможно, не более 17 операций на объект и т.п. Однако когда речь заходит о старых системах или о крупномасштабных компонентах, мы сталкиваемся со свершившимся фактом — система такова, какова она есть. Зачастую в конструкциях старых систем содержатся чрезвычайно крупномасштабные "объекты". Брокеры объектных запросов и компонентные модели специально создавались для многократного использования систем подобного типа. Вопрос состоит в том, как без помощи такого посредника можно воспользоваться преимуществами оболочки. Некоторые программисты обнаружили, что крупномасштабные компоненты естественным образом "вырастают" даже с учетом новых требований, и проследить такой рост в объектно-ориентированных моделях не всегда удается. Например, в работе [115] этот эффект описан в контексте программ геометрических преобразований изображений. Структуры данных могут быть простыми, однако описание обработки может оказаться слишком громоздким. Для таких систем наиболее естественным представлением оказались модели потоков данных. Это именно тот случай, когда кустарные оболочки могут стать даже более выигрышными, чем промышленные разработки.

В компонентной разработке также особо выделяются крупномасштабные компоненты, содержащие множество классов, реализующих различные интерфейсы. В этом случае любой компонент можно рассматривать как оболочку.

Такой подход к переходу — не единственный. Другие подходы подразумевают использование специализированных брокеров объектных запросов (ORB). Это хороший принцип, однако существует столько способов решения каждой проблемы, что можно сделать некоторые разумные обобщения. Специфические подходы применялись до тех пор, пока не появились программы среднего уровня, основанные на сообщениях, и продукты ORB. Сейчас для реализации оболочек старых систем довольно часто используют технологию ORB. Например, в одной крупной финансовой организации создана простая система взаимосвязи офисных продуктов на основе объектов, которые эффективно обобщают самоописательные пакеты данных. В результате был получен специфический брокер объектных запросов. При этом сам проект был завершен задолго до того, как на рынке появилась первая коммерческая система управления распределенными объектами на основе технологии CORBA. Более того, это приложение опередило спецификацию CORBA в смысле использования самоописательных данных, которые в настоящее время представляются с помощью XML.

Одной из самых больших проблем, связанных с концепцией объектных оболочек, является управление данными. Использовать оболочку не сложно до тех пор, пока не возникает потребность в разделении хранилища данных между старой базой данных и некоторыми новыми объектами.

4.4.2. Стратегии управления данными для оболочек

Подход, состоящий в использовании объектных оболочек, на первый взгляд выглядит идеальным, но при пристальном рассмотрении обнаруживается ряд проблем, связанных с управлением данными. Если при построении оболочки возникает необходимость в дублировании данных или их совместном использовании компонентами старой и новой систем, можно применить четыре возможные стратегии.

1. Обеспечьте дублирование действующей копии общих данных в обеих частях системы, а также актуальность этих копий. Проблема состоит в том, что требования к хранению могут дублироваться. Более того, существуют реальные вопросы совместимости, о которых следует позаботиться. Это нельзя назвать жизнеспособной стратегией для многократного использования или перевода на объектную технологию систем промышленного масштаба. Назовем такую стратегию **тандемом** или **рукопожатием** (handshake), так как она требует постоянной синхронизации обновлений и исправлений. Она будет работать только в том случае, если нет пересечений между данными старой и новой частей системы или их наложение невелико. Однако такое встречается крайне редко.
2. Хранение всех данных в старой системе и копирование их в новые объекты по требованию. Сообщения, адресованные старой системе, обеспечивают обработку изменений. Такую стратегию называют **кредитной** (borrowing) или **загрузочной** (download), так как данные временно заимствуются из оболочки. Это аналогично тому, что происходит во многих существующих приложениях MIS, где данные ночью загружаются из больших машин, а изменения передаются пакетом. Более тонкий вариант этой стратегии используется при связи объектно-ориентированных программ с реляционными базами данных. Данные из базы кэшируются в объектно-ориентированную

базу данных, вследствие чего программисты не обязаны сами заниматься отображением объектов на элементы базы, что повышает производительность (этот подход будет рассмотрен в главе 5). Этот вариант называется стратегией **кэширования** (caching) и зачастую реализуется в серверах приложений.

3. Скопируйте данные в новые объекты и разрешите данным старой системы "устаревать". Здесь тоже могут возникнуть проблемы совместимости. Оболочка будет обязана передавать сообщения, получать их и отвечать на них, что может значительно ее усложнить. Назовем это стратегией **захвата** (take-over), по аналогии с компанией, которая скупает акции другой.
4. Объедините последовательные фрагменты старой базы данных с соответствующими функциями. Это сложно и требует хороших методов объектно-ориентированного анализа, способных описать как старую, так и новую систему, а также требует средств преобразования документов проектирования исходной системы, таких как DFD. В результате это может стать самым многообещающим подходом к переходу на новую платформу. Эта стратегия называется **переводом** (translation), так как старое проектное решение переводится на язык объектной модели. Перевод легче всего реализовать, если старая система написана для набора важнейших данных с помощью метода пошагового усовершенствования (step-wise refinement). Такие структуры и использующие их программы естественным образом переходят в объекты новой системы. Улучшая эту стратегию, можно восстановить модель данных существующей системы и идентифицировать все операции доступа к данным в рамках этой модели. Примером может стать матричный подход CRUD. Метод CRUD часто используется для организации традиционных систем на основе необходимых структур данных. Эти вызовы впоследствии могут быть заменены новыми объектами, создаваемыми для этих элементов. Такая усовершенствованная версия называется **переводом на основе данных** (data-centred translation). Выполнимость этой стратегии зависит от сложности получения модели данных и сложности программ, заложенных в саму базу данных. Как отмечается в [657], при реализации этой стратегии могут оказаться полезными инструменты обратного проектирования, а также средства, обеспечивающие понимание семантики системы.

Выбор наиболее подходящей стратегии зависит от конечной цели, в качестве которой может выступать перевод системы на объектно-ориентированную реализацию, многократное использование компонентов, расширение системы, модификация или создание распределенного приложения переднего плана. Какая стратегия наиболее предпочтительна, если главной целью является переход от традиционной системы к объектно-ориентированной, а не просто многократное использование компонентов? Стратегия рукопожатия пригодна только для маленьких систем, и даже в этом случае будут существовать наложения между старыми и новыми компонентами. Кредитная стратегия может исказить старую систему. Она обычно не пригодна для перехода, кроме тех случаев, когда проводится четкое разграничение между существующими функциями и новыми требованиями (хотя эта стратегия и подходит для задач многократного использования старых компонентов). Эта стратегия не позволяет напрямую перемещать данные за пределы оболочки, а значит, придет время, когда потребуется предпринять гигантский скачок по переводу данных в новый формат, если для всех операций доступа к данным не будет использоваться СУБД. Эти стратегии, как и стратегии **перевода** систем, накладывают тяжелый отпечаток на всю идею построения оболочек. Для переноса

функций старой системы в новую подходят только две последние стратегии. При выборе стратегии нужно также учитывать тип системы, ее структуру, вид и качество документации. Изначально понятие оболочки касалось моделирования целостной системы средней сложности, ценность которой заключалась в самом коде, а не в управлении данными. Более того, основной задачей было многократное использование функций надежной системы, а не ее реконструкция.

Чаще всего приходится сталкиваться с четвертой стратегией. Она предполагает декомпозицию старой системы относительно логически связанных наборов данных и, если таковые существуют, использование некоторых существующих DFD для трансформации объектов на основе инкапсуляции. Если такие наборы не существуют, мы сталкиваемся с построением оболочки гораздо большей сложности на основе стратегии захвата. Это гораздо более дорогостоящий вариант.

Краеугольным камнем здесь является применение подхода, основанного на архитектуре, и использование шаблонов и наборов компонентов с хорошо определенными интерфейсами. В главе 7 речь пойдет о том, какой вклад в этот процесс вносят современные методы объектно-ориентированного проектирования, например Catalysis [204, 792].

4.4.3. Практические проблемы перехода

Если старые системы существуют одновременно в нескольких версиях, возникают проблемы другого типа. Например, коммерческий пакет для некоторой отрасли промышленности может быть адаптирован для нужд конкретных клиентов. Стоимость построения оболочки для каждой версии обычно очень велика. Подход, основанный на построении оболочек, будет работать только в том случае, если ядро системы — одно для всех версий. При этом придется модифицировать каждую версию оболочки отдельно. Такая ситуация сложилась в одном из проектов, в которых участвовал автор. Там существовало около 70 версий одного продукта, настроенных под конкретные узлы, разбросанные по всему миру. Во многих случаях конкретную версию обслуживала небольшая специализированная команда. Наряду с этим, декомпозиция существующей системы была чрезвычайно сложной из-за долгой истории модификаций. Была принята стратегия промоделировать систему, используя объектно-ориентированный анализ, а затем создать оболочку для ядра системы так, чтобы новые функции (компонент MIS) могли добавляться с использованием объектно-ориентированных методов. На первой стадии ядро системы должно было оставаться без изменений. Компьютер AS400, на которой работала система, вскоре должен был перейти на операционную систему UNIX, которая не поддерживает какие-либо объектно-ориентированные языки. Следовательно, в сжатые сроки пришлось написать новые объектно-ориентированные компоненты на традиционных языках. Чтобы убедиться в том, что новая система в будущем может стать полностью объектно-ориентированной, пришлось минимизировать затраты на этой стадии. Это привело нас к использованию объектно-ориентированного анализа и преобразованию его результатов в традиционный код программы. Пришлось использовать (в настоящее время несуществующий) продукт среднего уровня — NeWI, который позволял разработчикам трактовать системы как объектно-ориентированные, хотя они оставались написанными на C. Мы тоже создали объектно-ориентированное описание существующей системы, чтобы глубже ее понять и впоследствии отделить самостоятельные фрагменты для реализации стратегии перевода. Трактовка отдельных функций как объектов, а не как методов, оказалась полезной. Значительные усилия на первом этапе привели к разработке дополнительных функций и их интерфейсов с ядром существующей системы (с помощью оболочки).

Задачи перевода были перенесены на ближайшее будущее. Будет полезно расписать весь этот процесс по составляющим.

1. Построение оболочки для взаимодействия с новыми объектно-ориентированными компонентами на основе (наиболее вероятно) стратегии кредитования.
2. Выполнение объектно-ориентированного анализа старой системы.
3. Использование стратегии перевода (или перевода, основанного на данных).
4. Реализация брокеров объектных запросов.

В [337] показано, что оболочки хорошо работают в зрелых системах (в частности, в замороженных) в контексте требований сокращения объема обслуживания, который охарактеризован в этой работе как "чрезвычайно раздутый", хотя он всегда умышленно занижается. Основная мысль этой работы сводится к тому, что старые системы слабо структурированы, поэтому их понимание и построение оболочки стоят слишком дорого. Однако игра стоит свеч, если будет в достаточной мере сокращен потенциальный объем обслуживания. В работе рассматриваются сложные функции (анализаторы обычной грамматики), а не приложение, интенсивно работающее с данными.

Читатель может подумать, что лучшим подходом к переносу старых систем со сложными механизмами управления данными является построение оболочки, которая поддерживает объектно-ориентированные компоненты переднего плана, в рамках которых и реализуются необходимые новые функции. Стратегия тандема может использоваться только тогда, когда перекрытие невелико и система обслуживает отдельные базы данных. Исключение составляет случай, когда структура существующей системы уже основана на взаимосвязанных данных, что подразумевает стратегию перевода, или когда выигрыш от перевода системы настолько велик, что покрывает стоимость создания очень сложной оболочки согласно стратегии захвата. Для существующей СУБД (DBMS — Database Management System) можно построить оболочку, которая будет жизнеспособной долгое время, достаточное для последовательного перевода всех функций системы. После этого в какой-то момент все данные можно сразу перенести в объектно-ориентированную базу данных, исключающую необходимость в оболочке для традиционной базы данных. Существует отдельный вариант стратегии перевода, когда вся база данных представляет собой единую громадную "взаимосвязанную порцию данных". Это идеальный выбор для организаций, уже почувствовавших удовольствие от производительности реляционных баз данных.

Приняв решение построить оболочку и новые компоненты переднего плана, нужно выбрать средства для их создания. Некоторые производители, например Merant, предлагают продукты для создания оболочек для приложений на языке COBOL. Существуют также некоторые брокеры объектных запросов и средства среднего уровня, предназначенные для решения этого вопроса.

Ключевым вопросом, с которым сталкиваются многие IT-организации, является так называемая проблема "гуляша" (goulash). Она возникает в том случае, если в организации имеется набор несовместимого оборудования и программного обеспечения, который правдами и неправдами совместно использовали в течение многих лет. Концептуально этот "гуляш" можно рассматривать как систему, состоящую из больших компонентов, взаимодействующих посредством передачи сообщений с параметрами. Подход, основанный на построении оболочки, уместен только в том случае, если на объектно-ориентированную платформу нужно перевести лишь один из этих компонентов. Вместо создания оболочки для каждой старой

системы, которое будет, мягко говоря, дорогостоящим, лучше каким-то образом охватить оболочкой систему коммуникаций. Самым общим подходом к этой проблеме является использование брокеров объектных запросов и компонентное программирование. Некоторые средства EAI (Enterprise Application Integration) основаны на программах среднего уровня, объектно-ориентированных базах данных и технологии XML.

Упомянутый выше проект перевода был свернут по причине отсутствия подходящих программных средств. Исходя из этого, нашим главным инструментом будет сам объектно-ориентированный анализ.

Будем рассматривать не только задачу эволюции важных существующих систем, которые либо слишком велики, либо слишком сложны для того, чтобы просто их переписать, но и вопрос эволюционного перехода старых систем к объектно-ориентированной реализации. Это подразумевает необходимость многократного использования компонентов существующих систем (или даже целых пакетов) в новых или модернизированных объектно-ориентированных системах. Давайте обратимся к этому вопросу.

4.4.4. Многократное использование существующих компонентов и пакетов

Итак, мы рассмотрели стратегию построения оболочек для традиционных систем. Теперь займемся проблемой многократного использования существующих компонентов, когда нет прямой потребности или желания переводить их на объектно-ориентированный язык. В работе [230] показано, как можно обеспечить многократное использование высокооптимизированных алгоритмов или специализированных функций с применением стратегии оболочек. Это делается с помощью определения классов уровня приложения, методы которых вызывают подпрограммы старой системы. Старые системы могут быть облачены в оболочки группами или по отдельности. Последний метод обладает бо́льшим потенциалом для многократного использования. Последние усилия разработчиков ERP-систем по разбиению своих пакетов на компоненты снизили объем работы, необходимый для создания оболочек. Однако пользователям этих систем нужно перейти к использованию самой последней версии.

Перевод систем на новую платформу должен базироваться на существующих "пакетных" решениях. При этом можно построить оболочку, вызывающую подпрограммы пакета. В качестве альтернативы можно модифицировать сами пакеты и обеспечить экспорт данных, которыми будет управлять уже новая система, но это снова приведет к вопросу многократного использования функциональных компонентов старой системы. К тому же некоторые производители пакетов не готовы к поддержке таких изменений или даже не допускают их.

При построении оболочки приходится решать следующие проблемы [230].

- Разработчик не может выбрать лучшее представление для задачи в терминах объектов, так как оно в значительной мере предопределено старой системой. При этом существует вероятность, что оболочки будут представлять слишком крупномасштабные объекты, которые ограничивают возможность их многократного использования.
- Конструктор должен либо обеспечить функции и интерфейс старой системы для пользователя, либо защитить этого пользователя от возможных изменений старой системы. Очень сложно успешно реализовать и то, и другое одновременно. В общем случае это подразумевает предоставление доступа к старой системе только для чтения, что может помешать применению стратегии захвата для управления данными.

- Если старая система продолжает обслуживать данные, оболочка должна сохранять их состояние при вызове внутренней подпрограммы. Это препятствует применению стратегии перевода.
- Сборка мусора (garbage collection), управление памятью и уплотнение (compactification) (где это применимо) должны быть синхронизированы между старой системой и оболочкой.
- Необходимо поддерживать межсистемные инварианты, которые основаны на старых и новых наборах данных.
- Создание оболочки зачастую требует очень глубокого понимания старой системы. Это больше касается задач перехода, но остается в силе и для задач многократного использования.

Так как доступ к содержимому пакета на нужном уровне детализации вряд ли возможен, описанный выше подход, основанный на построении оболочки, обычно не применим. Лучше рассматривать пакет как фиксированный объект, предлагающий определенный набор служб, возможно в распределенной среде.

Если перевод на основе данных является лучшим подходом для преобразования и замены существующих систем (стратегия кредитования в этом случае не работает), то, когда речь идет о многократном использовании, самый жизнеспособный подход — именно кредитование. Если функциональность существующей системы или пакета интенсивно используется и ее поддержка требует разумных затрат, то новые функции можно реализовать отдельно на основе объектно-ориентированного подхода, а взаимодействие со старой системой можно осуществлять через оболочку. Оболочка используется для вызова служб старой системы и предоставления доступа к ее базе данных. Новые функции определяются как методы объектов, в которых инкапсулированы необходимые данные. Когда требуются данные, хранимые в старой системе, оболочке передается сообщение, и она вызывает соответствующую процедуру, возвращающую нужные данные. Аналогично выполняется обновление существующей базы данных, только в этом случае оболочка вызывает процедуру модификации данных.

На этом можно и закончить. С течением времени новая объектно-ориентированная система будет требовать все новой и новой функциональности, заменяющей и дублирующей части старой системы. В конце концов вместо заимствования части системы потребуется перевод на основе данных. Таким образом, поэтапная стратегия, рекомендованная в предыдущем разделе, может пригодиться для многих коммерческих систем.

Теперь можно подвести итог сказанному. Рассмотрим представленную ниже табл. 4.1. В этой таблице слово "Да" означает, что данная стратегия может заслуживать внимания при рассмотрении конкретного класса проблем, но это не значит, что она будет гарантированно работать. Слово "Нет" обозначает, что, скорее всего, такая стратегия не подойдет. Вопросительный знак говорит о том, что все зависит от обстоятельств. Стратегии, определенные в разделе 4.3.2, сравниваются по четырем возможным целям построения оболочек: переход сложной старой системы к новой объектно-ориентированной реализации, многократное использование компонентов без изменения ядра системы, расширение функциональности без изменения ядра и построение (возможно, распределенного) программного обеспечения переднего плана для реализации дополнительных функций. Заметим, что три последние цели очень близки между собой, двум последним соответствует одинаковое расположение значений "Да" и "Нет" в таблице.

Таблица 4.1. Пригодность стратегий перехода для различных целей

Стратегия	Цель			
	Миграция	**Повторное использование**	**Расширение**	**Построение ПО переднего плана**
Рукопожатие	Нет	Нет	Нет	Нет
Кредитование	Нет	Да	Да	Да
Захват	?	Нет	Нет	Нет
Перевод	?	Нет	Да	Да
Перевод на основе данных	Да	Нет	Да	Да

4.4.5. Использование объектно-ориентированного анализа

Рост интереса к объектной технологии в 1990-х годах привел к повсеместному использованию языков объектно-ориентированного программирования. Одновременно оказалось, что методы структурного программирования не только не поддерживают такие разработки, но и часто даже затрудняют их. Типичными приложениями объектной технологии стали графические пользовательские интерфейсы, распределенные вычисления, электронная коммерция и использование объектно-ориентированных и объектно-реляционных баз данных. Потребность в упорядочении подхода к таким разработкам привела к появлению методов объектно-ориентированного анализа. Еще одной областью применения таких методов были экспертные системы, в которых целесообразность объектно-ориентированного подхода определялась наличием структур наследования во фреймовых экспертных системах.

Для обеспечения взаимодействия компонентов объектно-ориентированных и традиционных систем зачастую требуется обратное проектирование старых систем с целью их описания в терминах согласованных сущностей. Ясно, что для этого требуются приемы объектно-ориентированного анализа, особенно при переходе систем к объектным технологиям. Таким образом, объектно-ориентированный анализ можно использовать как быстрый путь к объектно-ориентированному будущему.

Три источника появления объектов

Понятие объектов появилось в компьютерном мире по трем различным причинам. Сообщество разработчиков языков программирования создало объектно-ориентированную перспективу со своих позиций. Она включала моделирование, разработку графического пользовательского интерфейса и т.п. Теоретики баз данных испытывали потребность в абстрактных моделях представления семантики данных. Наиболее известной среди них стала модель Чена [159] "сущность-связь" или ER-модель (Entity-Relationship). Эти модели имели дело с абстрактными объектами и наследованием, но концентрировались на сущностях, относящихся к данным, игнорируя при этом процедурные сущности, которые тоже можно реализовать. Сообщество специалистов по искусственному интеллекту пошло на шаг дальше со своей концепцией фреймов (frame), определяющих сущности со своими атрибутами и процедурами.

Они могли использоваться для поиска отсутствующих значений и вызывать побочные эффекты. Однако фреймы оставались статичными структурами данных и не предлагали каких-либо средств для декларативного определения поведения сущностей. Процедуры были внешними по отношению к фреймам (хотя зачастую и связывались с ними) и не были инкапсулированы. Более того, фреймы гораздо ближе к прототипам, чем к объектам. Это значит, что не существует классов со своими экземплярами, которые наследуют все их свойства. Решение этой проблемы состоит в трактовке классов как определений прототипов (т.е. экземпляр может не иметь всех свойств (функций) класса, как в объектно-ориентированном программировании). Развивая эту мысль, можно сказать, что трехногая пьяная собака может быть экземпляром обычной собаки, которая имеет четыре ноги и, в основном, пьет воду. Такие языки, как SELF [767], основывались на прототипах, а не на классах. Это значит, что независимые от языка методы объектно-ориентированного анализа должны обеспечивать моделирование прототипов за счет перекрытия на уровне экземпляров. Если бы мы позволили себе такие вольности, то развитие объектной технологии обеспечило бы богатейший набор средств обхода препятствий при моделировании семантики данных, построении интеллектуальных систем и программировании. Давайте кратко рассмотрим каждую из этих перспектив.

Моделирование семантики данных

Эта задача возникает при разработке практических приложений баз данных и решении исследовательских проблем. Она лежит в основе концепции "клиент/сервер", триггеров баз данных и бизнес-правил, реализованных на сервере. Средства моделирования семантики данных широко используют правила и структуры наследования, но совершенно не затрагивают вопросы инкапсуляции. Заметно существенное взаимное влияние этой области и объектной технологии [341]. Задача моделирования семантики данных касается только самих данных и тесно связана с двумя другими областями исследований.

Объектно-ориентированное программирование

Объектно-ориентированный подход развивался под влиянием языка программирования Smalltalk и его среды разработки, созданной компанией Xerox в 1970-е годы прошлого века. Сегодня нас окружает множество объектно-ориентированных языков, среди которых C++, Eiffel и Java. Эти языки позволили решить множество программно-теоретических вопросов, таких как быстродействие, корректность, безопасность типов и выразительность. Как говорилось в главе 1, в объектно-ориентированном подходе под объектом понимается некоторая сущность, обладающая тремя свойствами: *уникальной идентичностью*, *инкапсуляцией* атрибутов и методов и *наследованием*.

Инкапсуляция означает, что объекты (классы или их экземпляры) обладают интерфейсом, определяемым их атрибутами и методами, а также скрытой реализацией своих структур данных и процессов. Ключевые преимущества этого подхода заключаются в том, что объекты являются неотъемлемой частью повторно используемого кода и построенные из них системы можно расширять за счет наследования и добавления новых или исключительных свойств.

Искусственный интеллект

Фрейм обычно описывается как структура данных с набором атрибутов, которые могут быть связаны с процедурами. Правда, эти процедуры не всегда хранятся в объекте, как в объектно-ориентированном программировании. Интересно, что этот подход был реализован

в большинстве объектно-ориентированных баз данных. Различие между фреймами и объектами состоит в отсутствии во фреймах инкапсуляции данных и процедур, скрытой за занавесью интерфейса. Кроме того, в системах искусственного интеллекта (AI — Artificial Intelligence) процедуры можно рассматривать как нефункциональные наборы правил, а не программы. Еще одним отличием между объектно-ориентированным подходом и AI является то, что в системах последнего обеспечивается богатый, по сравнению с объектными технологиями, выбор методов реализации наследования. Тем не менее схожесть перевешивает различия, и любой специалист, пожелавший вооружиться преимуществами объектной технологии (т.е. многократного использования и расширяемости), может продолжать использовать языки Nexpert и Smalltalk. В заключение можно сказать, что преимуществами AI являются более богатый набор функций наследования и способность представлять отношения в виде наборов правил, а недостатком — более низкий уровень инкапсуляции (а значит, многократного использования). Кроме того, методы AI менее эффективны при разработке пользовательских интерфейсов, создании CASE-средств и управлении существующими библиотеками классов или базами данных.

4.4.6. Объектно-ориентированный анализ и создание прототипов на основе знаний

Любой серьезный подход к объектно-ориентированному анализу должен быть независимым от языка. Недостаток этой идеи состоит в том, что спецификацию нельзя протестировать, а систему — собрать и использовать. Решением обеих проблем является создание прототипов. Прототип можно использовать для согласования спецификации с пользователями. Его даже можно частично внедрить в ожидании написания коммерческой версии программы на выбранном объектно-ориентированном языке. Однако система, построенная на языке прототипирования, должна быть обратима в том смысле, чтобы по ее коду можно было воспроизвести результаты аналитиков и проектировщиков. Аналитикам, работающим с бумагой, тоже необходим семантически богатый язык анализа, а также понятный и декларативный язык прототипирования.

Подобные средства появились сразу в двух областях IT, находящихся под влиянием искусственного интеллекта, — в экспертных системах и передовых базах данных. Для разработки экспертных систем были предложены семантически богатые, декларативные языки программирования. Расширения баз данных позволили программировать бизнес-правила с помощью продукционных правил и триггеров баз данных. Пока только Illustra и семантические базы данных предлагают нечто подобное. Даже объектно-ориентированные базы данных существенно не продвинулись в этом вопросе. Похоже вскоре ситуация изменится, и сообщество специалистов по базам данных осознает эту проблему. Кроме того, отдельные пользователи баз данных уже поняли, что бизнес-правила необходимо связывать с сущностями или объектами модели данных. В [34] отмечается следующее: "Четкое документирование бизнес-правил и связанных с ними объектов до начала программирования гарантирует правильное восприятие и понимание этих правил пользователем".

Можно заключить, что системы, основанные на знаниях, и расширенные продукты баз данных могут помочь в создании обратимых прототипов. Однако это лучше реализовать в сочетании с машинно-независимым анализом на основе семантически богатой, а значит обратимой системы обозначений.

Это хорошо согласуется с моим личным опытом разработки экспертной системы ARIES [135], которую мы строили для рабочих станций с намерением позднее перенести ее на персональные компьютеры. На каждой стадии, включая прототипирование, создавались независимые от реализации представления (проще говоря, бумажные модели). Это облегчило их перенос, поскольку переносился не сам программный код, а бумажная модель. Бумажная модель была обратимой, потому что она была написана на выразительном языке инженерии знаний, а не на необратимом языке программирования.

По мере того, как код экспертной системы совершенствовался и становился все более объектно-ориентированным, стало ясно, что системы, основанные на знаниях, являются хорошими средствами прототипирования благодаря своим выразительным средствам наследования. На этапе реализации мы с удовольствием отказались от них для повышения эффективности многократного использования, однако для анализа они были просто необходимы. Системы, основанные на знаниях, изначально обратимы, так как знания представлены в них в виде правил и объектов, а не программ или запутанных диаграмм. Использование такого подхода для создания спецификаций и прототипирования требует соответствующей системы обозначений, чтобы прототип экспертной системы можно было преобразовать в объектно-ориентированную бумажную модель или модель, построенную с помощью CASE-средства.

Важным вопросом для практиков остается поиск методов анализа и проектирования, которые поддерживают богатые возможности, заложенные в существующих системах. Еще один вопрос звучит так: "Как можно немедленно использовать доступные технологии?". Ответ на него сводится к следующему: использовать системы, основанные на знаниях, и средства прототипирования, строить свои разработки на семантически богатых методах анализа и проектирования. Возможно, на стадии реализации придется искать компромиссы из соображений производительности, порой даже используя COBOL. Однако поскольку большая часть средств, выделяемых на разработку, вкладывается не в реализацию, такой подход в будущем будет не только жизнеспособным, но и привлекательным для инвестиций.

4.4.7. Объектная технология как стратегия перехода

До сих пор я доказывал, что ключ к переходу на объектную технологию лежит в подходе к анализу и проектированию. Мы еще вернемся к этому вопросу. Теперь же я хочу вернуться к началу обсуждения и доказать, что ОТ — это не что иное, как системный анализ. Главный вопрос современного производства следующий: как обеспечить большую гибкость и продуктивность при одновременном сокращении затрат? Это включает в себя множество технических и культурных изменений, которые сложно правильно реализовать. Однако прошлый опыт привел меня к заключению, что сама ОТ, наряду с ясным видением и конструктивным руководством, является ответом на этот глас вопиющего в пустыне.

Мы анализируем системы, чтобы их было легче строить. Мы компьютеризируем производство, чтобы оно лучше функционировало. Хороший анализ является залогом хороших систем. Хорошие системы улучшают производственные показатели. Что же тогда представляет собой хорошая система? На этот вопрос нет однозначного ответа, однако я думаю, что сегодня вряд ли кто не согласится, что хорошая система, кроме всего прочего, экономна, открыта, гибка, устойчива, полезна и управляема. Какой вклад в создание таких систем может внести объектная технология? Экономичность, естественно, повысится за счет многократного использования, гибкости и управляемости можно добиться с помощью наследования и расширяемых интерфейсов. Однако остальные вопросы не столь прямо связаны с программированием объектов. Удобство системы сегодня связывается с наличием графического интерфейса.

Естественно, можно построить и совершенно неудобный графический интерфейс. Вероятно, самым важным вопросом любого современного производства является стоимость работы вычислительной инфраструктуры и затраты, связанные с изменениями в этой области. Существует большая надежда, что наступит время распределенных вычислений. Исходя из этого, передовые компании захотят перевести на эту платформу как можно больше своих приложений. К сожалению, на этом пути им придется столкнуться с большими трудностями. Их персонал не имеет для этого соответствующей подготовки. Распределенные вычисления куда сложнее обычных. Немногие в достаточной мере знают принципы построения пользовательского интерфейса, чтобы создать полноценный удобный GUI. К тому же еще и многократное использование... Даже эксперты в области объектной технологии спорят относительно способов реализации этого процесса и управления им.

Объектно-ориентированный подход позволяет описать распределенную природу систем вполне естественным способом, так как их объединяет передача сообщений. К тому же тот факт, что объектная технология предлагает способ управления сложностью посредством инкапсуляции и наследования, означает, по крайней мере, возможность представления сложной распределенной архитектуры.

Объектно-ориентированные средства разработки GUI упрощают создание таких систем и в то же время накладывают ограничения на стиль и непротиворечивость, которые, в свою очередь, способствуют удобству. Они значительно повышают продуктивность, могут использоваться для создания стандартов пользовательского интерфейса и обеспечивают многократное использование компонентов в этой области. Аналогично объектно-ориентированные средства создания экспертных систем становятся более общими, завоевывая авторитет схемами наследования, используемыми для снижения сложности систем правил, и обеспечивая дружественный и естественный диалог между пользователями и их приложениями.

Ясно, что опытные программисты, работающие на языках COBOL и RPG, *могут* освоить объектно-ориентированный подход, и некоторые компании доказывают это на практике.

По большей части остаются методологические и организационные проблемы. С моей точки зрения ключом к решению методологических проблем является хороший объектно-ориентированный анализ. Организационный вопрос остается ночным кошмаром.

Все изменения в организационном плане полны опасностей, и переход к объектной технологии не является исключением. Люди, как правило, всеми силами противятся изменениям, потому что не верят в преимущества перехода для себя или своей организации. Как отмечал Макиавелли [500]: "Ничто не связано с такими сложностями в организации, сомнениями в успехе и опасностями в реализации, как инициирование изменений". Тот, кто преуспел при старом порядке и боится потерять позиции, будет решительно возражать, а тот, кто может выиграть от изменений, будет предлагать лишь вялую поддержку. Потенциальные победители практически всегда "недоверчивы и никогда не верят в новшества, пока не проверят их на практике". Чтобы заручиться их поддержкой, нужно продемонстрировать успех на ранних стадиях. Более того, они должны быть уверены, что возврата к старому нет. Я всегда удивляюсь, как много времени и человеческих усилий было посвящено попыткам вернуться к хорошим техническим решениям, которые не выжили в технической гонке: Xerox Sigma, APL, CP/M 86, а теперь, возможно, даже и Mac. Как-то я работал в одной IT-организации, которая успешно преодолела путь к распределенным системам. Мы достигли этого с помощью позитивного отношения к объектной технологии, сильного руководства, ясного в*и*дения цели и усидчивости при изучении технологии. Я не собираюсь описывать историю этой организации. Это было бы слишком долго и неинтересно для широкой публики. Единственное, что я хочу предложить вниманию читателя — это ряд накопленных советов.

Нет смысла бояться. Цель стратегии перехода должна быть четко изложена в стиле боевой задачи. Она должна быть доведена до подчиненных соответствующими руководителями. В этом случае люди либо ее примут, либо начнут протестовать. Нужно сразу заявить, что большие машины в течение двух лет будут полностью заменены или что производительность должна вырасти в три раза. Эти амбициозные цели могут не вполне соответствовать действительности, однако покажут подчиненным направление движения. Поставленные цели должны быть связаны с технологией. Это позволит аргументировать, что многократное использование и объекты являются основой роста производительности. В заключение нужно выбрать определенную технологию. Вряд ли кто-нибудь может с уверенностью утверждать, что именно уцелеет и будет работать. Таким образом, вместо выбора одного языка или метода объектно-ориентированного анализа и проектирования OOA/D (Object Oriented Analysis/Design), нужно выбрать два или три, предоставляя руководителям проектов некоторую свободу в развитии своей деятельности. Обычно выбор языка определяет некоторый стиль программирования, так что неправильно будет остановиться на C++ и Ada. Можно запустить несколько различных проектов, что даст организации представление о наиболее подходящем языке, а также позволит извлечь уроки. Кроме того, руководителям проектов можно дать право выбора объектно-ориентированного метода проектирования. Однако с методом анализа так поступать нельзя, поскольку его строгое определение может обезопасить программу перехода к многократному использованию. Необходим подходящий метод, включающий определенный жизненный цикл и вдохновляющий на достижение требований, а также системный анализ и логическое проектирование. Начиная с самых ранних стадий выполнения проекта необходимо выделять объекты и помещать их в общее хранилище. Если выбранный метод не зависит от языка, эти объекты можно будет повторно использовать и реализовать на любом языке программирования. Представьте себе уныние разработчиков, если при смене языка всю библиотеку придется просто выбросить или даже частично переписать. С мнением о том, что мудрые компании должны выбирать стандарт UML (см. главу 6), я соглашусь, однако напомню, что система обозначений не является методом.

Важную роль играет и команда разработчиков. Главная проблема, с которой придется столкнуться большинству организаций, — это выбор между разработчиками, которые проработали в организации долгие годы и накопили громадный опыт, и новыми программистами, обладающими знаниями объектной технологии и современных методов. На самом деле, никакого выбора делать не нужно. Новые программисты должны обучать старых и осуществлять руководство. Более того, молодежь может прилежно перенимать опыт своих старших коллег. Этот процесс требует глубокого и прочного доверия между двумя группами разработчиков, и создать такое доверие — ключевая задача руководства. Необходимо ввести систему поощрений, которая будет стимулировать кооперацию между сотрудниками и повторное использование объектов, разработанных коллегами.

В заключение следует отметить, что необходимо отблагодарить и спонсоров разработки. Самой большой наградой для них будет сокращение затрат за счет повторного использования объектов и уменьшение сроков выхода на рынок. При этом повысится и качество самого продукта. Наследование сокращает затраты на поддержку и внесение изменений, отражающих изменения самого производства. Многократное использование положительно сказывается на финансовой стороне дела, однако его трудно реализовать. Многие разработчики готовы повторно использовать имеющиеся элементы на программном уровне, но этим процессом сложно управлять, поэтому все часто происходит неформально. Хочется верить, что планирование повторного использования начнется на уровне спецификаций.

4.5. Резюме

Объектная технология обеспечивает совместное использование ресурсов различными системами. В этой главе подчеркнута роль объектно-ориентированной модели в описании и понимании распределенных систем.

Существует три стратегии распределенных вычислений: централизованная, на основе репликации и разделения. Были рассмотрены некоторые типы распределенной компьютерной архитектуры: клиент/сервер, мультиклиент/мультисервер и сеть равноправных абонентов (peer-to-peer). Мы выделили четыре модели распределения: сервер баз данных, сервер транзакций, сеть равноправных абонентов и распределенное программное обеспечение переднего плана. Модель равноправных абонентов является самой общей и в то же время самой трудной для программирования и управления. Важным понятием в проектировании систем, распределенных или нет, является запаздывание (delay).

Существующие структурные методы внесли малый вклад и даже затормозили прогресс в развитии распределенных систем. Объектная технология предложила естественный путь моделирования распределенных приложений. Сетевые узлы в ней рассматриваются как абстрактные типы данных или объекты. Однако наследование и композиционные связи не могут распределяться по сети. Ассоциации в сети не могут быть минимизированы из соображений эффективности. Обычно системный уровень должен быть реализован на одном узле. Объектно-ориентированная модель подразумевает и требует прозрачности размещения.

Вычисления на платформе "клиент/сервер" означают разделение данных и процедур их обработки между несколькими клиентскими машинами, на которых запущены приложения, и единым сервером переднего плана, который предоставляет услуги всем клиентам. Технология клиент/сервер является особым типом распределенных вычислений. Объектно-ориентированные системы являются не чем иным, как системами мультиклиент/мультисервер (возможно, разделенными на уровни) или сетью равноправных абонентов. Многоуровневые модели более гибки, чем их двухуровневые предшественники. Технология RPC нарушает принцип прозрачности размещения, но это иногда может быть скрыто от пользователя сложным сетевым программным обеспечением. Кооперативная обработка является особым случаем распределенных вычислений, предлагающим равноправные соединения между серверами.

Системы баз данных часто организованы на основе концепции клиент/сервер. Реляционные системы, позволяющие применять хранимые процедуры, существенно сокращают сетевой трафик. Сложность состоит в том, что в их основу не заложена инкапсуляция и робастные методы объектно-ориентированного анализа. Объектно-ориентированные системы предоставляют эти возможности автоматически.

Брокеры объектных запросов и другие типы программных компонентов среднего уровня практически устраняют необходимость в сложных вызовах удаленных процедур (PRC). Приложения не обязательно должны быть написаны в объектно-ориентированном стиле, так как технология ORB предоставляет эффективную оболочку. Технология ORB привнесла преимущества объектной технологии в мир системной интеграции. Язык XML необходим для определения смысла данных, включаемых в сообщения. Однако одного языка для успешной совместной работы разрозненных систем не достаточно. Для этого также требуется общая бизнес-модель.

Ключевыми техническими свойствами современных систем являются распределенность, связность и гибкость. Эти факторы увели от железобетонного мышления и каскадного подхода к проектированию систем для больших машин к быстрой эволюционной разработке гибкого

распределенного программного обеспечения. В расширенном виде среда объектно-ориентированного программирования эффективна для прототипирования и описания мира параллелизма, распределенности и сообщений. Для объектно-ориентированной реализации необходимы объектно-ориентированный анализ и проектирование. Без хорошего объектно-ориентированного анализа и культуры управления проектами распределенное решение может вскоре стать более громоздким, чем старые системы.

Мы рассмотрели ряд стратегий для организаций, которые столкнулись с проблемами совместимости и перехода к объектной технологии, осветив все точки зрения программиста, который хочет разработать объектно-ориентированное приложение и использовать при этом функциональность приложений, написанных в традиционном стиле. Это привело к изучению стратегий объектных оболочек, объектных магистралей и систем "классной доски". Было показано, как заключить в оболочку приложение, существующее в нескольких различных версиях.

Самые большие затраты, связанные со старыми системами, приходятся на их поддержку. Однако желанию сразу перевести все системы на объектную технологию будут препятствовать экономические рамки. Необходимо основываться на существующих инвестициях и постепенно переходить на новую объектную технологию.

Первой рекомендованной стратегией перехода является построение оболочек. Создание объектных оболочек защищает инвестиции, вложенные в старые системы на этапе перехода к объектно-ориентированному программированию. Объектные оболочки представляют собой новую, объектно-ориентированную часть системы, которая призвана взаимодействовать с традиционными подпрограммами при получении сообщений от других компонентов. Создание оболочки подразумевает частичные инвестиции; после того как она заработает, практически все затраты на обслуживание прекращаются.

Внедрение оболочек — не такая простая задача, как это кажется на первый взгляд. Одной из проблем является крупномасштабность модулей. Обеспечить многократное использование гораздо легче для маленьких объектов, чем для больших. Имея дело со старыми системами, мы часто сталкиваемся с неподдающимися дроблению крупными "объектами". Брокеры объектных запросов специально предназначены для многократного использования такого рода "объектов".

Переход к ОТ на основе создания оболочки — не единственно возможный. Другие подходы предполагают использование брокеров объектных запросов, объектно-ориентированных баз данных и специализированных методов обработки.

Одной из самых больших проблем в концепции объектных оболочек является управление данными. Использование оболочки не представляет сложности, пока дело не доходит до разбиения данных между старым хранилищем и новыми объектами. Для решения проблемы можно применить четыре стратегии. **Стратегия рукопожатия** предполагает создание копии общих данных в обеих частях системы и поддержку их актуальности. Она работает только в том случае, если между данными старой и новой систем существует небольшое перекрытие или они вообще не пересекаются. **Стратегия кредитования** предполагает хранение всех данных в старой системе и их копирование в новые объекты (при необходимости). **Стратегия захвата** подразумевает копирование всех данных в новые объекты и "старение" данных старой системы. Здесь могут возникнуть проблемы совместимости. Оболочка должна будет принимать и передавать сообщения, а также отвечать на них, что существенно повышает ее сложность. **Стратегия перевода** связана с выделением взаимосвязанных порций информации из базы данных вместе со связанными с ними функциями. Эта стратегия требует наличия мощного метода объектно-ориентированного анализа, способного описать как старую, так

и новую систему. Однако в целом эта стратегия самая многообещающая. Более тонкая версия этой стратегии подразумевает выполнение обратного проектирования и извлечение модели данных из существующей системы, а также идентификацию всех операций доступа к данным в рамках этой модели. Это называется **переводом на основе данных**. Все эти стратегии можно применять в зависимости от цели перехода: замена старой системы, повторное использование компонентов, расширение системы или создание распределенного программного обеспечения переднего плана.

Если старая система существует и поддерживается во многих различных версиях, подход, связанный с построением оболочки, сработает только в том случае, если ядро системы остается единым для всех версий. Здесь нужно использовать поэтапный переход. Вначале создается оболочка для взаимодействия с новыми объектно-ориентированными компонентами на основе стратегии кредитования. После этого выполняется объектно-ориентированный анализ всей старой системы. В заключение для перехода к ОТ используется стратегия перевода (или перевода на основе данных).

Решение о создании оболочки или нового программного обеспечения переднего плана ставит задачу выбора средств реализации. Сегодня не существует специальных продуктов, реализующих стратегию оболочки, однако для решения задачи многократного использования предлагается множество средств, обеспечивающих реализацию ORB и GUI. Вместо построения оболочек для каждой старой системы иногда лучше заключить в оболочку систему взаимодействия между ними. Один из подходов к решению этой проблемы предполагает использование брокеров объектных запросов. Сегодня многие оконные среды содержат хорошие и удобные библиотеки классов GUI и средства для разработки объектных оболочек, так что можно использовать существующие программы создания интерфейса пользователя. Иногда главным средством может стать объектно-ориентированный анализ, но тогда потребуются хорошие CASE-средства.

Метод оболочек можно использовать для перехода системы к ОТ или для многократного использования существующих компонентов. Были перечислены проблемы, которые можно решить за счет построения таких оболочек. В табл. 4.1 показана применимость всех четырех стратегий в зависимости от целей перехода.

Мы увидели, что объектно-ориентированный анализ может стать основным плацдармом для выработки стратегии перехода к ОТ, и сама объектная технология также является стратегией. Остальные проблемы можно охарактеризовать как организационные и методологические.

Любое управление изменениями сопряжено с опасностью, и переход к объектной технологии не является исключением. Важную роль в реализации такого перехода играют стиль руководства и квалификация команды разработчиков. Большинство разработчиков готовы к повторному использованию чужих наработок на программном уровне, но этот вопрос с трудом поддается регулированию и часто решается на неформальном уровне. Поэтому многократное использование компонентов нужно предусматривать уже на уровне спецификаций.

4.6. Дополнительная литература

В [575] приводится расширенное и всестороннее (однако на сегодняшний день слегка устаревшее) введение в большинство вопросов разработки распределенных систем. В работе [211] обсуждаются вопросы совместного использования постоянных объектов, а также приводятся полезные комментарии по стратегиям блокировки и другим важным вопросам.

В [34] предлагается много полезной теоретической информации о программной инженерии, базах данных и распределенной обработке. Описанный метод применен к проектированию распределенной финансовой системы. В [53] можно найти подробное обсуждение вопроса применения объектно-ориентированного проектирования к моделированию компьютерных сетей. В [181] рассматриваются распределенные вычисления на техническом уровне, а также сетевые протоколы и работа механизма RPC.

Книга [8] — это сборник статей, затрагивающих многие сложные вопросы, связанные с параллелизмом объектно-ориентированных систем. Сентябрьский номер журнала за 1993 год *Communications of the ACM* посвящен параллельному объектно-ориентированному программированию и содержит несколько важных статей. В [549] рассматривается простая параллельная модель в контексте языка Eiffel.

Работы [574, 610] представляют хорошее введение (правда, слегка устаревшее) в технологию CORBA. В [573] обсуждаются шаблоны проектирования на основе технологии CORBA. Лучшим источником свежей информации остается Web-узел группы OMG (`www.omg.org`). В [391 приводятся начальные сведения о модели OMA. Источником исходной информации служит [722].

Работа [745] содержит прекрасный критический обзор компонентной технологии, существовавшей в конце 1990-х годов, а также детальное техническое описание и сравнение технологий CORBA и COM.

Работа [306] — мягкое введение в XML, сопровождаемое описаниями множества продуктов и приложений. Существует масса других книг по XML. Информацию по этой теме можно найти и в Web. В [111] содержится хорошее введение в XSL.

Работа [230] — это оригинальный первоисточник по объектным оболочкам, который хорошо дополняется работами [337, 657]. Некоторый полезный материал содержится в трудах конференции OOPLSA.

4.7. Упражнения

1. Укажите сходства и отличия между архитектурой клиент/сервер и объектно-ориентированным подходом.
2. Назовите преимущества централизованных и распределенных вычислений и укажите на их проблемы.
3. Что такое запаздывание? Как оно применимо к сетевому и архитектурному проектированию?
4. Дайте определение программ среднего уровня. Приведите примеры различных продуктов и их назначения.
5. Почему важна прозрачность размещения?
6. Что такое модель OMA?
7. В чем состоит различие между технологиями CORBA и MOM?
8. Опишите в деталях некоторый продукт CORBA.

9. Как называется стандарт OMG для межобъектного взаимодействия?
   а) Object Services Architecture
   б) Common Object Request Broker Architecture
   в) Object Interface Definition Standard
   г) иначе
10. Приведите определения терминов: опорный объект (stub), каркас (skeleton), объект-посредник, IDL и сортировка (marshalling).
11. Обсудите и сравните использование XML в электронной коммерции и любой другой отрасли, например в издательском деле.
12. Почему наличия XML и программ среднего уровня не достаточно для успешной интеграции приложений предприятия (технологии EAI)? Опишите проблемы, ожидающие типичный проект EAI.
13. Перечислите преимущества XML по сравнению с другими подходами к EAI.
14. Что означает буква S в аббревиатуре XSL?
15. Какое из понятий помогает перейти к объектной технологии, сохранив при этом инвестиции в существующий код?
   а) связывание
   б) объектная оболочка
   в) перегрузка
   г) зацепление
   д) полиморфизм
16. Что такое объектная оболочка? Как она может быть реализована?
17. "Объектная оболочка — оголтелый идеализм. Эта идея никогда не заработает на практике." Обсудите это высказывание.

5

Технология баз данных

У меня есть кот по имени Трэш... Если бы я попытался продать его (например, специалисту в области вычислительной техники), не стал бы подчеркивать его кротость по отношению к людям, его самодостаточность и способность питаться большей частью полевыми мышами. Скорее, я стал бы приводить доказательства, что он является объектно-ориентированным.

Роджер Кинг (Roger King). *My Cat is Object-Oriented*

В наши дни, чтобы продать программный продукт, необходимо рассказать миру, что он объектно-ориентированный или, возможно, компонентно-ориентированный. Давным-давно можно было сказать, что он реляционного типа. В этой главе рассматривается это смещение акцента или моды (если быть более циничным), а также приводится краткий обзор технологии реляционных и объектно-ориентированных баз данных.

В коммерческом мире реляционные базы данных не вызывали особых нареканий. Автор является одним из тех, кто в конце 70-х и начале 80-х годов был проповедником "реляционного Евангелия". В первой части этой главы будут рассмотрены причины появления реляционного подхода и некоторые его недостатки. Далее будет исследовано, каким образом многие из них исправлены в объектно-ориентированных базах данных. Для того чтобы перейти к этому обсуждению, вначале необходимо кое-что узнать о природе и истории баз данных, а также о моделировании данных вообще.

Все системы баз данных отличаются от других программ возможностями управления данными длительного хранения, а также эффективным и безопасным доступом к большим объемам этих данных. Давайте рассмотрим общие возможности систем баз данных, указанные Ульманом [765, 766].

- поддержка абстрактной модели данных
- поддержка высокоуровневого доступа или языка запросов
- поддержка управления транзакциями в многопользовательской среде

- поддержка управления доступом и принадлежностью данных
- поддержка проверки данных на правильность и непротиворечивость
- поддержка восстановления данных после сбоев системы и аппаратуры с минимизацией потери данных

В данной главе будут рассмотрены только первые две проблемы. Все остальные не будут исследоваться, поскольку, по сути, они относятся к предметной области систем баз данных. Некоторые из соответствующих ссылок даны в конце этой главы.

5.1. Краткая история моделей данных

Если не учитывать период развития вычислительной техники, когда программировали на коммутационной панели, то первым способом хранения данных можно считать применение последовательных файлов. Файл считывался от начала до тех пор, пока не были получены требуемые данные, а затем происходила перемотка ленты для приведения ее в готовность для следующего обращения. С появлением усовершенствованных запоминающих устройств, таких как барабаны и диски, языки программирования были расширены за счет включения в них операторов, предоставляющих возможность прямого доступа, иногда по непонятной причине называемого произвольным доступом (random access). Это вскоре привело к реализации, в которой скорость доступа была улучшена хэшированием (hashing) или сохранением индексных файлов.

Наличие индексных файлов привело к концептуализации структурной связи между этими файлами, отражающей часть структуры реального окружения или приложения. Поскольку принадлежность к классу является таким очевидным и повсеместно используемым структурным компонентом этого мира, файлы первых продуктов баз данных были связаны в иерархическую структуру. В системах для решения коммерческих задач это часто осуществляется в виде различных "вхождений" (occurrences) типов; например, компании с несколькими адресами. Иерархические базы данных, такие как IMS, были не только популярны, но и эффективны. Тем не менее многие коммерческие отношения не вписывались в чистые иерархии, и в результате исследований и системного анализа часто обнаруживались более общие сетевые связи. Это привело к разработке сетевых баз данных, примерами которых могут служить такие продукты, как TOTAL и IDMS. Эта концепция закреплена в стандартах комитета CODASYL. Сетевые системы стали почти такими же популярными и эффективными, как и их иерархические предшественники. Однако оба типа систем зависели от фиксированных указателей, и их было трудно изменить или расширить при реорганизации коммерческой деятельности. Все эти продукты разрабатывались в соответствии с практическими требованиями без использования преимуществ формальной теории. Не было детально разработанной "сетевой модели данных" или "иерархической модели данных". В этом отношении объектно-ориентированное программирование, на первый взгляд, повторяет историю баз данных в качестве *специального* (ad hoc) метода, которому не хватает формальной теории.

Следующим этапом было создание первой формальной модели данных — реляционной модели данных Тэда Кодда [175]. Эта модель базировалась на исчислении предикатов первого порядка (First Order Predicate Calculus — FOPC) и, оснащенная этой теоретической базой, поддерживала реляционно полный непроцедурный язык запросов. В настоящее время *de facto* установлен стандарт для определения данных и управления ими. Это язык

SQL, который, в основном, базируется на реляционной алгебре и является одним из возможных непроцедурных языков. Крайне избыточным синтаксисом язык SQL обязан как раннему языку GIS для IBM, так и реляционному исчислению, но его общепризнанным преимуществом является качество стандарта запросов и связи между базами данных. Это большое преимущество дополнено гибкостью реляционной модели и систем управления базами данных, основанных на этой модели. Атрибуты могут добавляться или удаляться без необходимости перестройки сложной системы указателей. Реляционные базы данных делают возможным переопределение систем, предназначенных для решения коммерческих и организационных задач, что позволяет достичь преимущества на мировом рынке в условиях жесткой конкуренции.

Вскоре стало ясно, что реляционная модель не является единственно возможной формальной моделью данных, и другие модели тоже могут иметь определенные преимущества.

В общем, модель данных является математическим формализмом, состоящим из системы обозначений для описания данных и структур данных (информации), а также набора соответствующих операций, которые используются для управления этими данными или, в крайнем случае, маркерами (token), представляющими эти данные. Функциональная модель Шипмана [706] является одной из наиболее успешных по теоретическим показателям, в то время как модель "сущность-связь" (Entity-Relational Model) Чена (Chen) — это одна из широко используемых коммерческих моделей, хотя ей и недостает согласованной теории, описывающей операции над данными. Идеи функциональной модели тесно связаны с идеями семантических сетей в искусственном интеллекте (Artificial Intelligence — AI) и с функциональным программированием. Таким образом, базовая логика ближе к лямбда-исчислению (lambda-calculus) Черча-Карри (Church-Curry), чем к исчислению предикатов первого порядка. Эта модель используется в некоторых языках, в частности Daplex и Adaplex. В языке Daplex две сущности с одинаковыми значениями компонентов могут, тем не менее, отличаться наличием отдельных ссылок. Считается, что они имеют "объектную идентичность" (object identity). Этого нельзя добиться в стандартной реляционной модели и ее расширениях, предложенных Коддом и другими. В расширениях предлагаются уникальные идентификаторы для кортежей, позволяющие реализовать выражения, обозначающие уникальный объект, который существует отдельно от своих компонентов. Это связано с концепцией "вызова по ссылке" (call by reference), которая в языках программирования является альтернативой "вызову по значению" (call by value).

Функциональный подход, реализованный в языке Lisp, не предполагает различий между данными и функциями (или программами) и допускает абстрактные типы данных, т.е. типы определяются неявно с помощью используемых операций. Например, тип `List` может быть определен с помощью операций `CAR`, `CDR` и `CONS`. Новые типы могут быть сгенерированы с помощью обобщения (generalization) (IsA или AKO) и агрегирования (aggregation) таким образом, чтобы сущности могли быть организованы в иерархии или сети. Следовательно, функциональная модель обеспечивает перспективу для новых разработок, особенно в связи с появлением аппаратных средств с параллельной и потоковой архитектурой, когда ресурсоемкость неразрушающего присваивания снижается.

Различные расширенные реляционные модели, начиная с модели Чена "сущность-связь" (Entity-Relationship — ER), были предложены для преодоления недостатков реляционной модели. Главным среди этих недостатков, по мнению автора, является трудность включения семантики предметной области в реляционную реализацию. Стало ясно, что семантические модели должны включать как функциональную семантику, так и семантику данных. Для представления данных предлагались также двоичная реляционная модель, где все отношения имеют только по два атрибута, и модель RM/T Кодда (для более подробного ознакомления с

обеими моделями обратитесь к книге [215]). Это были попытки борьбы с общепризнанными недостатками реляционной модели. Длительное время модели "сущность-связь" (ER-модели) рассматривались только в качестве инструментов для анализа и проектирования. Но в последнее время предпринято несколько попыток создания законченных СУБД, базирующихся на ER-модели или даже дедуктивной модели, обеспечивающей интегрирование функциональной семантики (бизнес-правил) в код. Например, абстрактная машина Fact, которая базируется на расширении двоичной реляционной модели, была реализована в программном продукте Generis, который будет рассматриваться ниже в разделе 5.3.

Разработка формальных моделей данных привела к коммерческой разработке реляционных систем управления базами данных (РСУБД), что, в свою очередь, содействовало дальнейшему развитию этих моделей. В настоящее время почти все системы баз данных, для которых производительность не критична, разрабатываются как реляционные системы. В крупномасштабных системах, требующих высокой эффективности, например в современных системах резервирования авиабилетов, иерархические и сетевые системы продолжают существовать наряду с новыми объектно-ориентированными системами управления базами данных (ООСУБД).

Оказалось, что функциональная модель очень точно соответствует сетевым базам данных и обеспечивает такую же рационализацию. Выяснилось, что IsA-связи и функции непосредственно соответствуют связям и "наборам" (sets), реализуемым в базах данных CODASYL.

В настоящее время разработчики проявляют интерес к системам с объектно-ориентированными базами данных. В [765–766] определена предварительная объектно-ориентированная модель данных, с помощью которой можно представлять иерархические и сетевые базы данных. Прежде чем приступать к ее рассмотрению, необходимо более подробно исследовать реляционную и предшествующие ей модели.

5.1.1. Недостатки ранних баз данных

Ранние иерархические и сетевые базы данных, главным образом, страдали от отсутствия гибкости. В иерархической модели можно отобразить только определенные типы структурных отношений, а именно одиночное наследование или выделение подтипов. Сетевые системы предоставляют возможность построения более общих графов, но по завершении разработки системы эти отношения очень трудно изменить. Добавление и удаление атрибутов также часто предполагает большую доработку. В этом разделе приводится часть теории баз данных и кратко рассматривается сущность этих ранних подходов.

Простейшей непримитивной структурой данных можно считать список, развитием которого являются списки списков и списки с древовидной структурой, реализованные в таких языках, как Lisp. Каждому, кто писал компьютерную программу, приходилось открывать файл или поток данных и считывать его запись за записью, пока не выполнится некоторое условие. Эта процедура аналогична просмотру списка или списка списков, если в файле хранится несколько видов символов. Например, рассмотрим компьютеризированный телефонный справочник, в котором хранятся фамилии, адреса и телефонные номера некоторой группы населения. Если известны имя и адрес, по ним легко отыскать номер. Но гораздо сложнее отыскать имя, если известны адрес и номер телефона. Это связано с тем, что списки и файлы имеют логическую структуру, и информация в них отсортирована в определенном порядке. Для облегчения задачи поиска имени необходимо отсортировать файл в другом порядке. Телефонная книга представляет собой список списков, который можно рассматривать как таблицу. Ситуация может усложняться в случае списка списков, которые в

свою очередь являются списками списков, и т.д. Например, телефонная книга может иметь структуру, показанную на рис. 5.1.

В отделении по связям с общественностью компании ABC имеется три добавочных номера, поэтому необходимо включить в справочник еще одну группу записей, содержащую список добавочных номеров и контактных имен. Несложно заметить, что эта информация имеет древовидную или иерархическую логическую структуру. Использование повторяющихся групп информации в ранних базах данных явилось следствием ограничений, накладываемых последовательными запоминающими устройствами в ранних вычислительных машинах, такими как ленты.

Phone_No	*Company*	*Dept*	*No_Of_Exts*	*Extension*	*Contact*
999-8888	Aardvark	Sales	1	102	S. Jones
777-1234	ABC Ltd	PR	3	110	J. Doe
				111	A.N. Other
				133	E. Codd
123-1234	Blue Inc	Mkting	1	2001	P.C. Plodd

Рис. 5.1. Файл с повторяющейся группой

Следующий уровень сложности возникает при наличии нескольких файлов. При использовании технологии прямого доступа в приведенном выше примере нет необходимости хранить повторяющиеся группы в одном и том же файле. Можно определить базу данных так, чтобы в файле компании содержались просто логические указатели на файл добавочных номеров. Тогда структура будет выглядеть приблизительно так, как показано на рис. 5.2.

ФАЙЛ 1

Phone_No	*Company*	*Dept*	*Dept_Code*
999-8888	Aardvark	Sales	17
777-1234	ABC Ltd	PR	23
123-1234	Blue Inc	Mkting	24

ФАЙЛ 2

Dept_Code	*Extension*	*Contact*
17	102	S. Jones
23	110	J. Doe
23	111	A.N. Other
23	133	E. Codd
24	2001	P.C. Plodd

Рис. 5.2. Декомпозиция для устранения повторяющихся групп

Рассмотрим два файла с общим полем, которое их связывает. Логически эта связь может быть представлена иначе. На логическом уровне это можно описать как отношение "один ко многим" между отделениями и добавочными номерами. В этой схеме добавочный номер не может находиться в двух отделениях, что выявляет одно из основных ограничений иерархического подхода к структуре данных — трудность организации отношения "многие ко многим". Несложно представить себе и более сложные иерархии.

Сообразительный читатель скажет, что автор сильно упростил пример, поскольку одинаковые коды отделений могут иметь две различные компании. Строго говоря, во втором файле ключ должен быть интерпретирован как код отделения компании. Впрочем, идея должна быть понятна, несмотря на это упрощение.

Для того чтобы почувствовать сложность сетевой модели, предлагается рассмотреть базу данных SACIS, предназначенную для хранения информации об отдельных товарах и поставщиках, которая показана на рис. 5.3. Эти отношения могут быть описаны утверждением: "у каждого товара может быть много поставщиков и у каждого поставщика может быть много товаров". Для пункта i4 текущий поставщик не указан. Символ * определяет конец связи "многие". Более подробная сетевая диаграмма показана на рис. 5.4.

INo	*IName*	*Description*
i1	Marble	S" glass

*

SNo	*SName*	*Location*	*Price*
s1	ABC	London	.30
s2	Aardvark	Paris	.30

INo	*IName*	*Description*	
i2	Bat	Willow	

*

SNo	*SName*	*Location*	*Price*
s1	ABC	London	20
s2	Aardvark	Paris	40
s3	Blue	New York	20

INo	*IName*	*Description*
i3	Kite	2 string

*

SNo	*SName*	*Location*	*Price*
s1	ABC	London	40

INo	*IName*	*Description*
i4	Ball	Rubber

Рис. 5.3. Данные, представляющие отношение `supplied-by` (поставляется)

Обратите внимание на кольцевую структуру связей и на то, что существует два пути поиска ответа на запрос: "Найти описание группы товаров i2, сделанное компанией s2". Можно начать с компании и просматривать указатели на отдельные товары, входящие в группу, или начать с товара и искать компанию. При этом неизвестно, какая из стратегий будет наиболее эффективной.

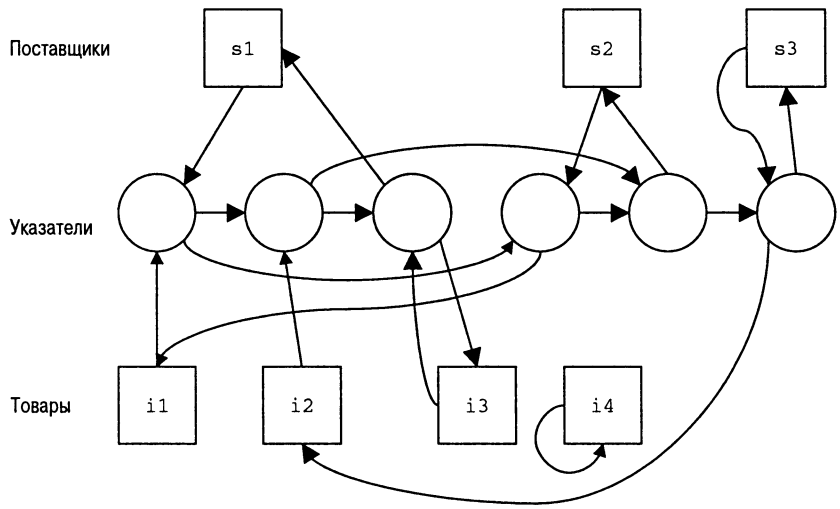


Рис. 5.4. Структура сетевых указателей для отношения `supplied-by` (поставляется), представленного на рис. 5.3

Впервые увидев диаграмму, подобную представленной на рис. 5.4, читатель может решить, что она выглядит пугающе запутанной. Затраты на сопровождение действующих сетевых баз данных подтверждают эту точку зрения. Тем не менее доступ с использованием указателей очень эффективен.

Как будет видно из дальнейшего, в реляционных базах данных преодолены некоторые ограничения иерархических и сетевых баз данных, связанные с показателями гибкости.

5.1.2. Реляционная модель и ее преимущества

Вследствие коммерческой важности реляционной модели данных, необходимо кратко рассмотреть ее основные концепции и теоретическую базу.

Создание реляционной модели данных было мотивировано несколькими причинами. Среди них было желание использовать формальные методы проектирования баз данных, обработки запросов к ней и ее модификации. Кроме того, хотелось иметь возможность проверки корректности программ, базирующихся на непроцедурных описаниях, и было непреодолимое желание удовлетворять принципу "бритвы Оккама", который сводится к тому, что теория должна быть максимально простой при сохранении ее выразительной мощи. Основная идея реляционной модели заключается в том, что данные представляются в виде набора таблиц или "плоских файлов" (flat files). В базе данных не допускаются никакие повторяющиеся группы или неявные иерархии, а также никакие жесткие структурные связи. Логические

отношения между данными строятся во время выполнения или содержатся в таблицах. Таким образом, один и тот же тип объекта используется для представления как сущностей, так и отношений. Эти таблицы "перекрестных ссылок" могут перестраиваться без необходимости реорганизации основных данных. Это является большим преимуществом в базах данных, для которых модели предметной области подвергаются значительным организационным изменениям. Поразительно, что даже в настоящее время в большинстве методов разработки систем основное внимание все еще уделяется выделению наборов данных или сущностей, хотя данные могут со временем изменяться. В реляционной базе данных предполагается, что все данные, в принципе, могут со временем изменяться до тех пор, пока не поступит некоторое указание извне. Поэтому реляционная модель изначально была неправильно понята как ее сторонниками, так и противниками. Главным источником непонимания является путаница между логическими и физическими моделями данных. Реляционная модель — это логическая модель данных. Для иерархической или сетевой структуры реально не существует никакой логической модели. Эти структуры являются физическими. В сущности, программные продукты реляционных баз данных для повышения эффективности могут быть реализованы в виде сетевых баз данных. Логическая реляционная модель дает пользователям возможность представлять одну структуру данных множеством различных способов, посредством так называемых пользовательских представлений. Таким образом, одним из важных преимуществ реляционных баз данных является более высокая степень их удобства для пользователя.

До сих пор реляционные базы данных были представлены неформально. Это зачастую ведет к неправильным представлениям, поэтому теперь необходимо перейти к более точному описанию. Читатель, уже имеющий полное представление о реляционной теории или, возможно, имеющий богатый практический опыт разработки баз данных и питающий отвращение даже к несложным математическим формулам, может смело перейти к разделу 5.3 или бегло просмотреть оставшуюся часть этого раздела.

Реляционная модель состоит из двух внутренних и двух внешних частей. Первая внутренняя часть — это структурная часть, в которой используются понятия *доменов*, *n-арных отношений*, *атрибутов*, *кортежей*, а также *первичных* и *внешних ключей*. Вторая — это управляющая часть, основными инструментальными средствами которой являются реляционная алгебра и/или реляционное исчисление и реляционное присваивание. Внешние части включают раздел целостности как сущностей (первая внешняя часть), так и ссылок (вторая). Вторая внешняя часть, которая относится к реляционной модели, — это метод проектирования, включающий теорию нормальных форм. Часть этой терминологии будет объяснена далее, по мере необходимости.

В математическом смысле *отношение* (relation) представляет собой некое подмножество декартова произведения множеств. Для списка множеств A_1, ... A_n декартово произведение является множеством всех списков или множеств с повторяющимися n элементами множеств A_j (bags), в котором из каждого A_j берется только по одному элементу. Такое множество называется упорядоченным n-кортежем или просто **кортежем** (tuple). Отношение называют **n-арным** (n-ary), если оно содержит n атрибутов. **Доменом** A_j называется множество возможных значений атрибута, а **атрибутом** — метка этого множества. Отношение эквивалентно понятию таблицы, которое чаще всего используется в компьютерном контексте (рис. 5.5).

---

[1] A bag — список, в котором элементы могут повторяться (множество с повторяющимися элементами), как противоположность обычному множеству, где повторение элементов не допускается.

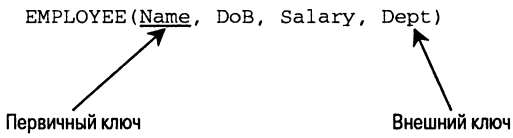


Рис. 5.5, а. Отношение с четырьмя атрибутами, одним первичным ключом и одним внешним. Внешний ключ `Dept` является первичным ключом для других отношений, представляющих отделы

Name	DoB	Salary	Dept
R.L. Stevenson	1/1/1850	10,000	Shipping
C. Dickens	25/12/1812	4,000	Curiosities

Рис. 5.5, б. Некоторые кортежи отношения `Employee`

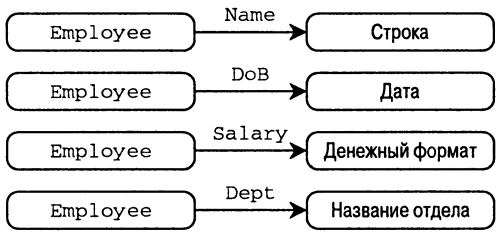


Рис. 5.5, в. Домены четырех атрибутов

Теперь рассмотрим связи (relationships) между отношениями базы данных. Они, конечно, тоже являются отношениями. Чтобы их различать, Чен [159] использует терминологию "сущность-связь" (entity-relation) и "связь-отношение" (relationship-relation). Здесь же, для краткости, будем говорить о связях (links), не подразумевая при этом соответствия со связями (links) базы данных CODASYL. Оба вида связей должны соответствовать некоторым ограничениям целостности. Для каждой сущности из набора атрибутов должен быть определен, по крайней мере, один первичный ключ, который однозначно идентифицирует каждый кортеж в произвольный момент времени. Кроме того, он должен быть представлен в первой нормальной форме, т.е. значения атрибутов не могут иметь сложную структуру (содержать повторяющиеся группы, списки и т.д.), а должны быть данными атомарных типов (числами, строками и т.д.). Это не сложно обеспечить для приложений обработки данных, но при обработке списков, хранении текстов и образов, а также при работе со знаниями могут понадобиться соответствующие расширения. Связи характеризуются двумя свойствами: кратностью и модальностью. **Кратность** (multiplicity) связи (иногда называемая **кардинальностью** (cardinality)) может принимать значения "один к одному" или "многие к одному". Иногда для описания связей используется свойство **модальность** (modality), определяющее необходимость или возможность данной связи. Этот уровень структуры добавляется к реляционной модели и, в строгом смысле, не является ее частью. Ограничения целостности, кратности и модальности обычно задаются в коде приложения и почти всегда реализуются на некотором внешнем процедурном языке. Таким образом, эту часть теории можно рассматривать как *расширенный* реляционный анализ (ERA — Extended Relational Analysis).

Как видно из рис. 5.6, существует 16 возможных типов связи, соединяющих две сущности или два отношения. Для удобства, обозначения, применяемые в анализе ERA, показаны слева, а их эквиваленты на языке UML — справа. Направление связи определяет способ ее интерпретации — в зависимости от направления читается только вторая пара значков модальности/кратности. На рисунке внешние символы обозначают кратность, а внутренние — модальность. Для описания кратности используются следующие обозначения: "вороньи лапки" ("многие"), поперечная черта ("один") и отсутствие обозначения ("нуль"). Значение "нуль" используется редко, поэтому оно не включено в 16 возможных вариантов, иначе их было бы 36. Символом модальности для обозначения возможности является O, (читается, как "может быть" или иногда более грубо "является необязательным"). Символом для обозначения необходимости является поперечная черта или 1 (читается, как "должно быть"). Таким образом, на рис. 5.6, *а* связь сверху вниз интерпретируется так: "каждый служащий должен быть связан только с одним отделом", а связь снизу вверх — "в каждом отделе может быть много служащих". В UML с целью объединения двух представлений используются числовые обозначения (символ "звездочка" (*) заменяется на n). Так что `0..*` обозначает необязательную связь "один ко многим". Эта система обозначений очень похожа на систему обозначений для ER-модели, предложенную Ченом (Chen). Она рассматривается ниже в этой же главе. Для анализа данных существует несколько других систем обозначений с той же семантикой. Например, в системе обозначений CASE*METHOD в Oracle [55], которая основана на идеях Бэчмана (Bachman), для обозначения возможности используется пунктирная линия.

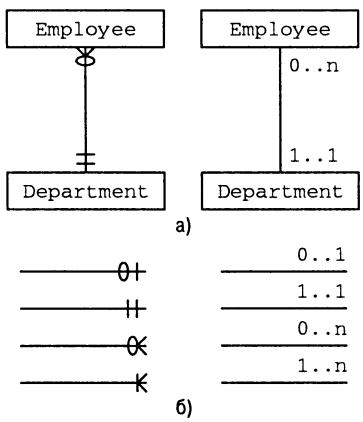


Рис. 5.6. Обозначения, применяемые в ERA (слева), и их эквиваленты в языке UML (справа): а — примеры диаграмм; б — четыре варианта кардинальности ассоциации

Необходимо также определить внешний ключ, т.е. атрибут, который является первичным ключом некоторого другого отношения. Правила целостности определяют, что происходит со связанными отношениями, когда таблица подвергается операциям модификации или удаления.

Теперь необходимо обратиться к той части модели, которая связана с управлением, т.е. к средствам, с помощью которых могут быть выражены запросы и требования модификации. По сути, существуют два метода и различные их комбинации. Эти два метода именуются

реляционным исчислением и реляционной алгеброй. Считают, что язык управления является *реляционно полным*, если с помощью одиночного (и заведомо непроцедурного) оператора этого языка может быть задана любая возможная операция над базой данных. Операционное определение реляционного исчисления обусловливается именно этим. Язык, допускающий задание всех возможных операций над базой данных с помощью нескольких операторов (а значит, процедурный язык), называется реляционной алгеброй. Первым методом, выделенным в основополагающем труде Кодда, было реляционное исчисление, которое является языком для поиска и модификации, основанным на подмножестве исчисления предикатов первого порядка. Позже был разработан основанный на этом язык QUEL, используемый в системе Ingres.

В реляционном исчислении поиск выполняется с использованием переменной кортежа, которая может принимать значения из некоторого заданного отношения. Выражение исчисления кортежей определяется рекурсивно, как формула исчисления предикатов, сформированная из переменных кортежа, реляционных операторов, логических операторов и кванторов. В этом смысле реляционное исчисление является непроцедурным. Ниже приведен пример оператора исчисления предикатов.

```
(∃x,y) (t∈ITEMS) ∧ x=INAME ∧ y=DESCRIPTION ∧ INAME='Kite'
```

Данный оператор можно рассматривать как запрос, возвращающий имя и описание всех элементов с именем 'Kite'. Следует обратить внимание на то, что переменная кортежа, `t`, является свободной, а связанные переменные, `x` и `y`, соответствуют атрибутам. Приведем синтаксис этого запроса на языке QUEL.

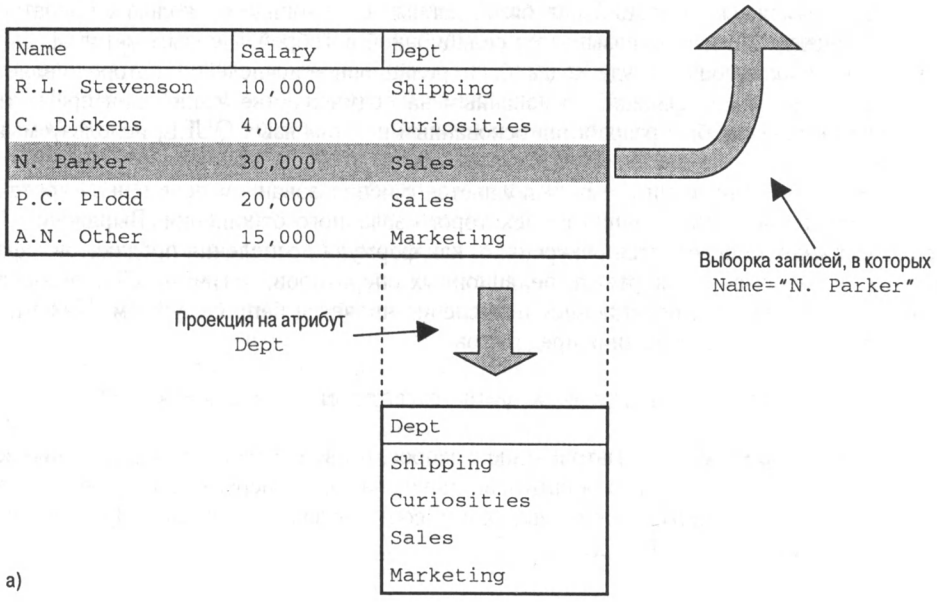
```
RANGE OF T IS ITEMS
RETRIEVE(INAME.DESCRIPTION)
WHERE INAME='Kite'
```

Подобный синтаксис позволяет описывать в этом непроцедурном стиле операции произвольной сложности.

При альтернативном подходе запросы и модификации должны рассматриваться, как выражения из последовательности алгебраических действий. Это ближе к процедурному стилю и очень близко к физической реализации, так как результаты будут зависеть от порядка оценивания. В конкретных реализациях оптимизатор запроса обычно пытается выбрать оптимальный порядок оценивания на основе прозрачности ссылок алгебры. Реляционная алгебра основана на четырех примитивных операциях, а именно: выборке (selection), проекции (projection), объединении (union) и соединении (join). Эти действия иллюстрируются на рис. 5.7. Результатом выборки по предикату являются те кортежи, которые удовлетворяют этому предикату. В рамках теории множеств это можно описать так: $\{x: p(x)\}$[2]. Эту операцию выборки можно рассматривать, как операцию выбора подмножества по горизонтали. Соответствующая операция выбора подмножества по вертикали — это внешняя проекция декартова произведения. Если две таблицы имеют одинаковые атрибуты, их объединение может быть сформировано с помощью конкатенации и удаления каких бы то ни было дубликатов в первичном ключе. Соединение двух отношений A и B через реляционный оператор (или

[2] Читается так: множество всех элементов x, таких, что результат выполнения оператора p от x равняется истине.

Name	Salary	Dept
N. Parker	30,000	Sales



а)

Employees

Name	Dept
R.L. Stevenson	Shipping
C. Dickens	Curiosities
N. Parker	Sales
A.N. Other	Marketing

Managers

Name	Dept
R.L. Stevenson	Shipping
F. Bloggs	Sales

Name	Dept
R.L. Stevenson	Shipping
C. Dickens	Curiosities
F. Bloggs	Sales
N. Parker	Sales
A.N. Other	Marketing

б)

EMPLOYEES

Name	Salary	Dept
R.L. Stevenson	10,000	Shipping
T. Smolett	11,000	Shipping
L. Sterne	9,000	Shipping
I. Graham	13,000	IT
C. Dickens	4,000	Curiosities

MANAGERS

Name	DoB	Dept
R.L. Stevenson	1/1/1850	Shipping
C. Dickens	25/12/1812	Curiosities
N. Parker	13/3/1956	Sales
F. Bloggs	8/11/1977	Sales
A.N. Other	29/2/1948	Marketing

в)

EMP.Name	DoB	Salary	Dept	MAN.Name
C. Dickens	25/12/1812	4,000	Curiosities	C. Dickens

Рис. 5.7. Операции реляционной алгебры: а — выборка и проекция; б — объединение двух (совместимых) отношений; в — соединение двух отношений по предикату Employee.Dept=Manager.Dept. Обратите внимание на то, что кортежи, для которых предикат принимает значение null, удалены

двухместный предикат) p — это объединение всех кортежей, которые являются результатом конкатенации кортежа из отношения A с кортежем из отношения B, определенные атрибуты которых соответствуют оператору p. В данном случае дубликаты также удаляются. Производные операции реляционной алгебры, такие как вычитание (difference) и пересечение (intersection), рассматриваться не будут. Данное изложение реляционной алгебры для краткости несколько упрощено. Для получения более полных сведений читатель может обратиться к работам [214, 255]. Реляционную алгебру можно рассматривать в качестве средства, определяющего области действия для таких операций, как поиск, обновление, формирование представлений, авторизация и т.д. Чисто алгебраические языки на практике используются чрезвычайно редко. В [338] приводится синтаксис такого языка, а именно языка ASTRID.

Существует несколько гибридных языков, частично базирующихся на исчислении и алгебре. Наиболее известным является язык SQL, основанный на языке System-R компании IBM. Причиной разработки SQL была необходимость создания "структурированного" языка. Позже разработчики пришли к заключению, что в него необходимо (или удобно) включить значительную часть реляционной алгебры и исчисления, причем настолько большую, что язык SQL стал крайне избыточным и, следовательно, неизящным. Тем не

менее он стал фактически промышленным стандартом, и поэтому ниже будет представлен его синтаксис и использование.

Приведенный выше запрос на языке SQL выглядит следующим образом.

```
SELECT INAME, DESCRIPTION
FROM ITEMS
WHERE INAME='Kite'
```

Не следует путать оператор выбора SELECT с определенной выше операцией выборки (selection) из реляционной алгебры. Этот запрос можно рассматривать следующим образом: спроецировать отношение ITEMS на INAME и DESCRIPTION и выбрать кортежи, соответствующие предикату INAME ='Kite'.

Обратите внимание на то, что реляционные языки запросов оперируют таблицами или множествами кортежей. Запрос всегда возвращает таблицу. Строго говоря, к такому языку необходимо относиться, как к кортежно-реляционному исчислению. Можно также построить доменно-реляционные исчисления, где переменные пробегают значения из доменов атрибутов. В кортежном исчислении переменные соответствуют кортежам. Например, рассмотренный выше запрос на языке доменного исчисления можно выразить следующим образом.

```
RETRIEVE x, y FROM ITEMS(p, 'Kite', q, r, y)
```

Здесь предполагается следующая схема отношения

INo	IName	SNo	Price	Description

и все переменные рассматриваются как свободные. Обратите внимание на позиционный стиль и сходство с языком запросов по образцу Query By Example компании IBM. Фактически этот язык базируется на доменном исчислении.

Было упомянуто, что отношения должны быть заданы в первой нормальной форме. На самом деле это условие располагается в нижней части иерархии нормальных форм (рис. 5.8), а наиболее важными являются третья нормальная форма и третья нормальная форма Бойса-Кодда (Boyce-Codd). Теория нормальных форм — это просто способ формализации понятия "хороший проект" с точки зрения здравого смысла.

Чтобы облегчить определение различных нормальных форм, необходимо выяснить, что же означает функциональная зависимость одного атрибута отношения от другого. Атрибут является **функционально зависимым** (functionally dependent) от другого тогда и только тогда, когда каждое значение второго атрибута единственным образом определяет значение первого атрибута. Другими словами, проекция отношения на эти два атрибута является функцией (т.е. однозначным, всюду определенным отношением).

Функциональная зависимость не симметрична. Например, если отношение определяет множество всех точек параболы, то значения по оси Y функционально зависят от значений по оси X, но *не наоборот* (рис. 5.9). Рассмотрим более конкретный пример. Для большинства компаний внутренний номер телефона будет функционально зависеть от городского номера компании. В [214] отмечено (и автору данной книги это кажется интересным), что функциональная зависимость — это специальная форма ограничения целостности. Другими словами, функциональная зависимость является условием законности, связанным с семантикой ситуации.

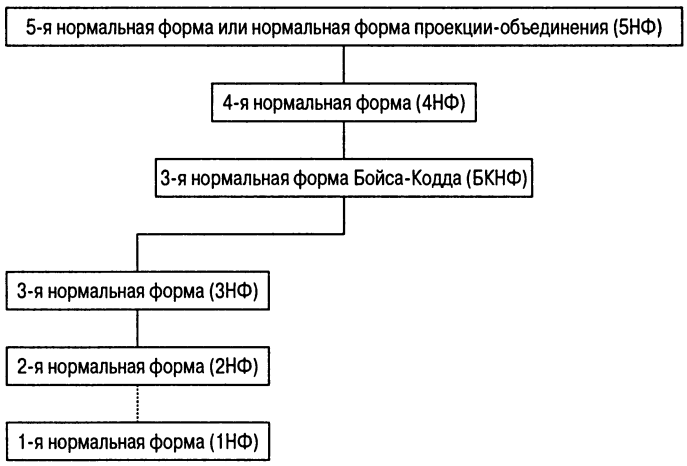


Рис. 5.8. Иерархия нормальных форм

Функциональные зависимости выражают конкретные связи в реальном мире и требуют понимания предметной области приложения. Функциональные зависимости, вообще говоря, не могут быть обнаружены автоматически, а только опытным системным аналитиком.

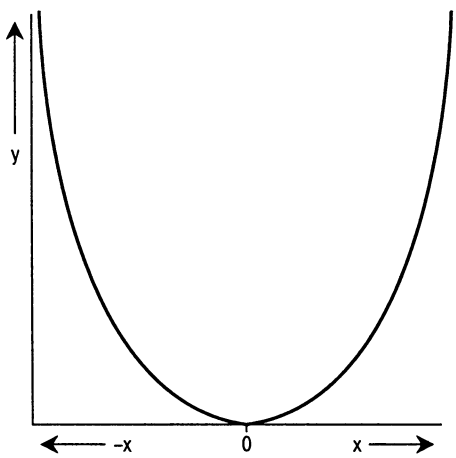


Рис. 5.9. Отношение на множестве вещественных чисел, определенное параболой ($y=x^2$)

Интуитивно понятно, что отношение находится в (третьей) нормальной форме тогда и только тогда, когда первичный ключ однозначно идентифицирует кортеж и другие атрибуты независимы друг от друга и функционально зависимы от ключа. Эти атрибуты являются реквизитами или описаниями сущности. Уже упоминалось, что для сущности, описанной в третьей нормальной форме, значение атрибута зависит от "ключа, полностью от ключа и только от ключа".

Отношение находится в **первой нормальной форме** тогда и только тогда, когда все атрибуты могут принимать только атомарные значения. При этом обычно используются числовые или строковые значения, но в действительности это зависит от лежащего в основе логического языка. Отношение находится во **второй нормальной форме** тогда и только тогда, когда, помимо всего прочего, каждый неключевой атрибут функционально зависит от первичного ключа, а не от конкретного подмножества значений этого ключа. Проектные решения, построенные с учетом вторых нормальных форм, преодолевают некоторые аномалии, касающиеся операций вставки, удаления или модификации кортежей. В качестве примера можно рассмотреть следующие пять отношений, показанные на рис. 5.10.

Отношение 1

SNo	Location	Currency	INo	Prise
1	London	GBP	1	30
1	London	GBP	2	20
1	London	GBP	3	40
1	London	GBP	4	20
1	London	GBP	5	10
1	London	GBP	6	10
2	Paris	FF	1	30
2	Paris	FF	2	40
3	Paris	FF	2	20
4	London	GBP	2	20
4	London	GBP	4	30
4	London	GBP	5	40

Отношение 2

SNo	Location	Currency
1	London	GBP
2	Paris	FF
3	Paris	FF
4	London	GBP
5	New York	USD

Отношение 3

SNo	INo	Prise
1	1	30
1	2	20
1	3	40
1	4	20
1	5	10
1	6	10
2	1	30
2	2	40
3	2	20
4	2	20
4	4	30
4	5	40

Отношение 4

SNo	Location
1	London
2	Paris
3	Paris
4	London
5	New York

Отношение 5

Location	Currency
London	GBP
Paris	FF
New York	USD

Рис. 5.10. Нормализация отношений

В отношении `Отношение 1` (рис. 5.10) факт существования поставщика невозможно зафиксировать с помощью его номера (`SNo`) до тех пор, пока он не обеспечит поставку единицы товара (`item`), поскольку ключевое поле `INo` не может принимать пустые значения (`null`). Более того, при удалении девятого кортежа теряется вся информация относительно поставщика `3`. Это явление часто называют "ловушкой связности" (connectivity trap). Чтобы отредактировать местоположение (`location`) поставщика 1, необходимо модифицировать

все кортежи, связанные с этим поставщиком, иначе база данных станет противоречивой. Эту ошибку легко допустить на практике. В отношении `Relation 1` рассматриваемая база данных представлена в первой нормальной форме, но не во второй. В результате разбиения этого отношения на отношения `Relation 2` и `Relation 3` достигается вторая нормальная форма, позволяющая преодолеть все эти проблемы. Заметим, кстати, что при втором способе представления экономится память, необходимая для хранения базы данных.

Обратите внимание на неявные функциональные зависимости в этой базе данных: атрибут `Currency` (денежная единица) зависит от `Location` (размещение), который зависит от `SNo` (номер поставщика), а `Price` (цена) зависит как от поставщика, так и от номера единицы товара, которые вместе составляют первичный ключ отношения `Relation 1`, а по отдельности — первичные ключи отношений `Relation 2` и `Relation 3`. Значит, и в этом новом проектном решении не все проблемы преодолены. Невозможно зафиксировать денежную единицу по факту размещения, пока в базу данных не будет введен поставщик с данным местоположением. Аналогично при удалении последнего кортежа отношения `Relation 2` теряется информация о долларах. Так как эта таблица все еще содержит некоторую избыточность, остаются и проблемы, связанные с модификацией. Для того чтобы избавиться от этих проблем, необходимо обратить внимание на то, что функциональная зависимость атрибута `Currency` от `SNo` возникает в результате появления других зависимостей. Такая зависимость называется **транзитивной** (transitive dependency). Если же реорганизовать базу данных и представить ее в виде совокупности отношений `Relation 3`, `Relation 4` и `Relation 5`, то удаляются все транзитивные зависимости и преодолеваются все упомянутые выше аномалии.

Отношение имеет **третью нормальную форму** тогда и только тогда, когда оно находится во второй нормальной форме и каждый неключевой атрибут нетранзитивно зависит от первичного ключа. Существует более строгая версия этого определения, сформулированная Бойсом (Boyce) и Коддом (Codd). Она сводится к следующему. Отношение находится в **третьей нормальной форме Бойса-Кодда** тогда и только тогда, когда каждый атрибут, от которого зависят другие атрибуты, является потенциальным ключом (candidate key) (т.е. может быть первичным ключом). Это определение не сформулировано в терминах первой и второй нормальных форм и, таким образом, немного изящнее. Можно показать, что это определение подразумевает все нормальные формы, которые были определены до этого момента, с возможным исключением определения первой нормальной формы [764].

Обратите внимание на то, что соединение отношений `Relation 4` и `Relation 5` восстанавливает отношение `Relation 2`, а соединение отношений `Relation 2` и `Relation 3` восстанавливает отношение `Relation 1` без потери какой-либо информации (не вводятся никакие ложные кортежи). Это не обязательно может иметь место, хотя можно показать, что любое отношение допускает такое разбиение без потерь. При проектировании базы данных и управлении ею должна быть проявлена осторожность, чтобы избежать операций проекции, сопровождаемых соединением, при котором таким образом теряется информация. Например, проецирование отношения `Relation 2` по атрибутам `(SNo, CURRENCY)` и `(LOCATION, CURRENCY)` привело бы к соединению "с потерями" по `CURRENCY`.

Чтобы преодолеть эту проблему соединения, сопровождаемую потерями, и другие небольшие проблемы, вводятся четвертые и пятые нормальные формы. Их точное определение довольно сложное и в этом тексте будет неуместным. Для ознакомления с этими определениями читатель может обратиться к работам, которые приведены в разделе библиографии в этой главе. Неформальное определение дано в следующем абзаце.

Сначала подытожим приведенные выше описания нормальных форм. Для первой нормальной формы, или 1НФ, настоятельно требуется, чтобы вводимые значения атрибутов были "атомарными". Иными словами, не должно быть никаких "повторяющихся групп" или списков, а значения должны быть скорее примитивными типами данных, чем указателями. Это означает, что значения *не* могут представлять сложные объекты с их собственной структурой. Считается, что во второй нормальной форме (2НФ) кортежи однозначно определяются первичным ключом. Обратите внимание на то, что это не означает уникальную идентичность объектов, потому что две записи с одинаковыми значениями атрибутов, но с различной идентичностью не могут быть сохранены. В реляционных системах идентичность объекта моделируется путем введения уникальных полей идентификации, например `EmployeeNo` (номер служащего). Считается, что в наиболее полезной форме 3НФ (НФ Бойса-Кодда) каждый детерминант является потенциальным ключом. Другими словами, каждая строка атрибутов, которая уникально идентифицирует кортеж, может быть ключом. Автор рассматривает эту нормальную форму как своего рода институт здравого смысла, поскольку именно так от начала до конца описал бы отношения опытный разработчик. 4НФ помогает избежать избыточности, 5НФ предотвращает соединения с потерями, т.е. при соединении двух (или более) отношений и последующем их разбиении на исходные формы не теряются никакие данные.

Теория нормальных форм является одним из аспектов восходящего проектирования (bottom-up design) и дополнением к методам нисходящего проектирования (top-down design). Строго говоря, она не является частью реляционной модели, так же как не является частью модели и вся теория целостности.

Одним из существенных недостатков обычного реляционного подхода относительно создания бизнес-приложений является неестественный способ обработки отношений, которые непосредственно воспринимаются в многомерной форме. Лучшим примером этого является область финансового моделирования и организации хранилищ данных. Отказ признать в качестве атомарного какой-либо объект более высокого уровня (список, вектор и т.д.) усложняет решение относительно тривиальных задач моделирования крупноформатных электронных таблиц в рамках реляционной модели. Легкость использования таких программных пакетов, как Lotus 123, Excel и Oracle Express, объясняется богатством структур данных и их хорошим соответствием способу человеческого восприятия. На практике большинство организаций хранит необработанные данные в реляционных таблицах и агрегирует их различными способами перед передачей результатов в такие пакеты для моделирования и в приложения по поддержке принятия решений. Создается впечатление, будто реляционная модель оперирует большими массивами необработанных данных, но это не является очевидным фактом. Некоторые приложения по поддержке принятия решений очень высокого уровня полностью соответствуют реляционной организации данных.

В пределах реляционной модели эта проблема может быть частично решена путем перехода из кортежного исчисления в доменное. Пример такого подхода может быть найден в системе для финансового моделирования Oracle Express. В литературе по Oracle Express эта система рассматривается как программный продукт, поддерживающий реляционную модель данных. Это справедливо в том смысле, что модель данных является логической и в ней поддерживаются алгебраические операции, такие как перемещение, проекция и т.д. Однако версия реляционной модели, представленная в этой системе, существенным образом отличается от модели, описанной выше. Она удовлетворяет определенным практическим потребностям и устраняет недостатки классической модели. Язык запросов в этой системе базируется на доменном исчислении.

Система Express, наряду с несколькими другими постреляционными базами данных и системами поддержки принятия решений, такими как Cache, предоставляет пользователю логическую модель данных, которая отличается от всех сетевых, иерархических и реляционных представлений. Это многомерная модель данных, которая связана с многомерными переменными, область определения которых можно представить в виде гиперкуба. Например, в финансовой модели размерностями соответствующего шестимерного гиперкуба могут быть время, статьи баланса, компании, регионы, продукция и денежные единицы. Нет необходимости представлять эти размерности в виде реальных направлений, как в декартовой геометрии, поскольку они сами могут быть множествами. В действительности это представление данных является простым преобразованием реляционной модели, где значения по каждой размерности эквивалентны значениям атрибутов, выбранным из домена.

Преимущество представления данных в системе Express заключается, конечно же, в синтаксической легкости, с которой можно выразить операции над проекциями отношений. Типичным запросом этого вида может быть следующий: "Перечислите все бухгалтерские балансы компаний в регионе А".

Оба вида реляционных систем обладают большой мощностью. Они позволяют легко вносить изменения в структуры данных и защищают пользователей от сложности непроцедурных языков запросов, которые могут быть автоматически оптимизированы. Постепенно преодолеваются проблемы производительности. По прошествии периода первоначального неприятия, в наши дни реляционные базы настолько широко применяются в промышленности, что большинство разработчиков систем уже даже не рассматривают иерархические или сетевые решения. Они используются только для приложений с интенсивными транзакциями, которые упоминались выше. Корпорация IBM рекомендует модель DB2 даже для больших приложений, и перед большинством проектировщиков систем стоит только задача выбора соответствующего программного продукта среди известных систем: Informix, Ingres, Oracle или Sybase. Критерием для принятия решения являются только такие факторы, как поддержка, переносимость или даже маркетинговые мероприятия.

5.1.3. Семантические модели и методы анализа данных

Как уже говорилось, модель данных является математическим формализмом, состоящим из системы обозначений для описания данных и структур данных и множества адекватных операций, которые используются для управления этими данными. Модели данных также помогают разработчику устранить избыточность данных в запоминающем устройстве и противоречивость в структуре проекта.

Существует несколько расширенных моделей данных и связанных с ними систем обозначений (нотаций). К ним относится система обозначений ERA, разработанная в рамках традиций Чена (Chen) и уже упомянутая в этой главе. Эти модели с широкими возможностями относятся к **семантическим моделям данных** (semantic data models).

Модели, основанные на логике предикатов (в том числе реляционная модель, поддерживающая декларативный язык доступа), отличаются от общего декларативного стиля других моделей. Такие модели являются ориентированными на значение (value-oriented) и не поддерживают объектную идентичность. Еще один класс моделей данных можно рассматривать как объектно-ориентированный в том смысле, что поддерживается объектная идентичность и, как правило, наследование. Эти "объектно-ориентированные" модели данных включают неявные модели старых иерархических и сетевых баз данных. В [765, 766] приводятся рассуждения о преимуществах и недостатках декларативности и объектной идентичности.

В моделях, базирующихся на отношениях и только отношениях, тип результата операции всегда одинаков (таблица) и, следовательно, может быть включен в некоторую другую операцию. Таким образом, операции можно легко компоновать. Это несущественно для моделей, которые поддерживают абстрактные типы данных, поскольку объектная идентичность приводит к вариациям типов, а результатом операции может стать совершенно новый тип. Таким новым типам могут быть необходимы совершенно новые операции, определенные в соответствии с этими типами, хотя в объектно-ориентированной системе они легко могут быть унаследованы.

Семантические модели данных берут начало с двоичной реляционной модели [3]. Схемы двоичной модели — это, по существу, семантические сети, приведенные к классам. Система обозначений Чена [159] изначально создавалась как язык проектирования. Она кардинально отличается от реляционной модели, в которой были сделаны попытки разделить физическое и логическое описание данных и разработать мощные непроцедурные языки запросов. Семантические модели данных были предназначены для моделирования связей и целостности. Современные семантические модели также включают очень сложные средства реализации наследования, структур, инстанцирования (instantiation) и выделения подтипов (subtyping). Общая идея состоит в том, чтобы иметь возможность моделировать данные на высоких и низких уровнях абстракции и зафиксировать в этой модели как можно больше смысла.

В [718] показано, что реляционная модель была недостаточно выразительна, чтобы охватить всю ER-модель. Кроме того, чисто реляционные базы данных не могут поддерживать сложные объекты любого вида, включая абстрактные типы данных. Это запрещается первой нормальной формой (1НФ). Объектно-ориентированное проектирование позволяет вводить столько типов, сколько необходимо, и обеспечивает реализацию наследования. Таким образом, для облегчения реализации проекта необходимо использовать объектно-ориентированные базы данных.

Наиболее широко используемыми семантическими моделями данных являются модели типа "сущность-связь" или ER-модели (Entity-Relationship). Различные ER-модели возникли в результате попыток анализировать и моделировать данные независимо от программного обеспечения базы данных, которое будет использоваться для реализации проекта. Основные принципы построения этих моделей и соответствующие системы обозначений были разработаны в [47, 159]. ER-модели преследуют две цели: классифицировать отношения по типам и реализовать бизнес-правила, в том числе правила обеспечения целостности ссылочных данных. Чен предложил два типа ER-моделей: "сущность-отношение" и "связь-отношение". В работе [686] предложено пять типов моделей, но и это не окончательная цифра. Имеется несколько семантически более богатых расширений базовой ER-модели, причем часть из них носит одинаковые имена. Расширенная модель "сущность-связь" EER (Extended Entity-Relationship Model), представленная в [754], включает обозначения, в которых обобщение (generalization) и иерархии подмножеств различаются. Еще одна расширенная модель "сущность-связь" (тоже EER), описанная в [255], представляет собой расширенный набор решений из большинства других ER-проектов. Для достижения своих целей авторы этих EER-моделей были вынуждены ввести в них классы или абстрактные типы почти в стиле объектно-ориентированного подхода. Возможно, это означает, что понятие абстрактных классов с наследованием является каноническим, т.е. своего рода "самым лучшим" способом рассмотрения понятий высшего порядка относительно сущностей и концепций в контексте моделирования данных.

На рис. 5.11 показана типичная система обозначений для EER-модели. Обратите внимание на то, что тернарные связи могут быть факторизованы с помощью атрибутов, соответствующих третьей сущности. Например, можно создать сущность с именем `project assignments` (распределение проектов) и атрибутами: `Employee` (Служащий), `Project` (Проект) и `Location` (Расположение).

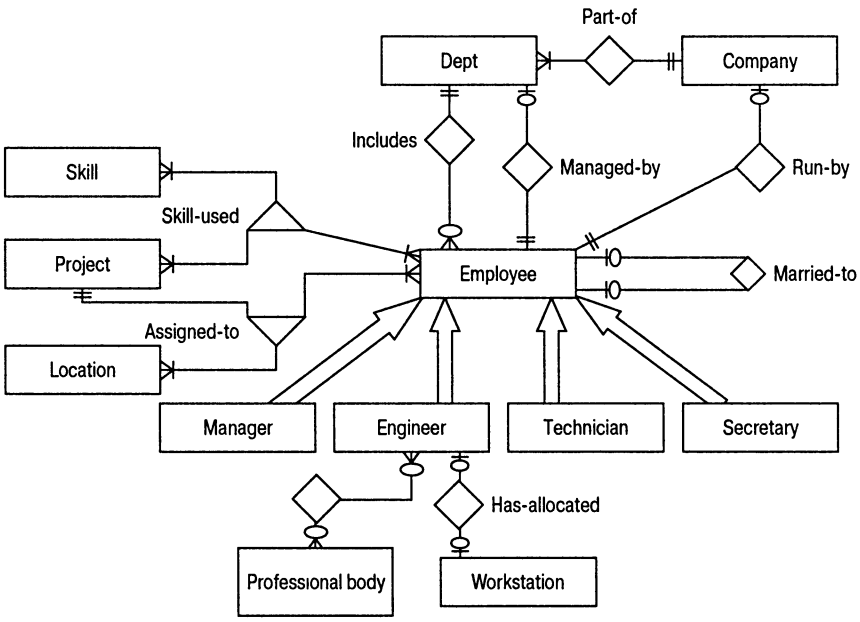


Рис. 5.11. Типичная расширенная ER-модель данных, в которой показаны бинарные и тернарные связи и наследование (широкие стрелки)

Стоит заметить, что в ER-модели связи могут быть объектами первого класса, поскольку они сами могут иметь атрибуты. На рис. 5.12 показан пример, где для связи `uses-car` (использует—автомобиль) необходим атрибут, показывающий промежуток времени, в течение которого конкретный автомобиль использовался конкретным служащим.

Фактически в ER-моделях возможны три способа представления связи. Объект может содержать атрибут, домен которого является другим объектом, представляющим связь; оба объекта могут содержать атрибуты, являющиеся друг для друга доменами; или может существовать конкретный объект связи. Например, на рис. 5.12 существующие атрибуты объекта `Employee` можно заменить атрибутом `CarUsed` или объектом `Car` с атрибутом `UsedBy`. Причиной выбора третьего представления является наличие у связи нескольких важных атрибутов.

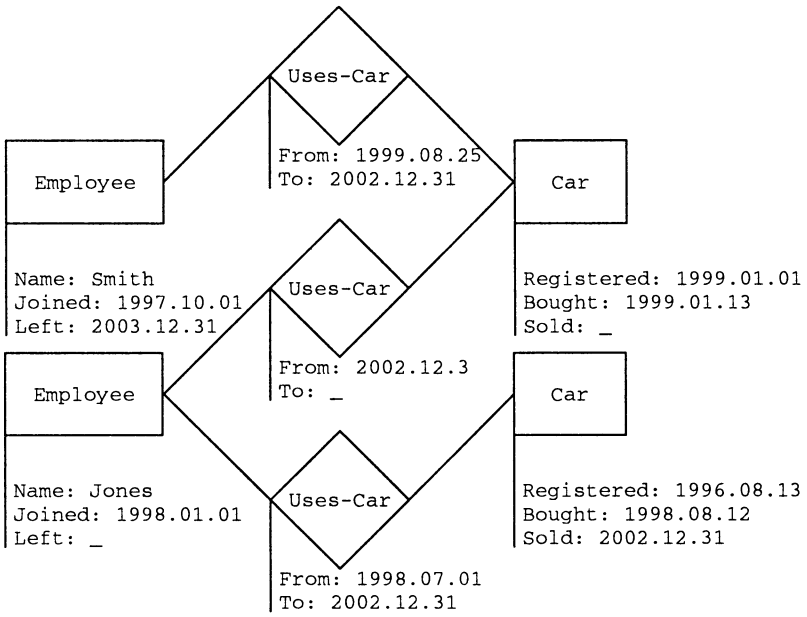


Рис. 5.12. Экземпляры связей с атрибутами в ER-модели

Системы обозначений в стиле Чена (Chen) и Бахмана (Bachman) широко используются; при этом обозначения могут изменяться. Одним из наиболее интересных аспектов некоторых версий системы обозначений Чена является их явное обращение к семантике наследования и другим структурам. В частности, связи подтипов (subtype relationships) делятся на четыре вида.

E/M — исключающий/обязательный (Exclusive/Mandatory)

E/O — исключающий/необязательный (Exclusive/Optional)

I/M — включающий/обязательный (Inclusive/Mandatory)

I/O — включающий/необязательный (Inclusive/Optional)

Термин "обязательный" обозначает, что член супертипа должен принадлежать одному из подтипов, т.е. подтипы составляют исчерпывающий список или *разделы* типа. Термин "исключающий" означает, что пересечение (intersection) каждого подтипа с другими подтипами пусто. Термин "необязательный" говорит о том, что список не является исчерпывающим. Могут существовать и другие, пока еще неопределенные подтипы. Термин "включающий" обозначает, что подтипы могут перекрываться. Обозначения для них показаны на рис. 5.13.

В традиционных ER-моделях чрезвычайно трудно или даже невозможно управлять такими объектами баз данных систем автоматизированного проектирования CAD/CAM, как изображения, геометрические рисунки или структуры с произвольным текстом. Автор полагает, что именно по практическим соображениям разработчики моделей данных стремятся к объектно-ориентированным моделям данных, и по аналогичной причине поставщики баз данных вынуждены вводить объектно-ориентированные конструкции в свои, по сути дела, реляционные программные продукты.

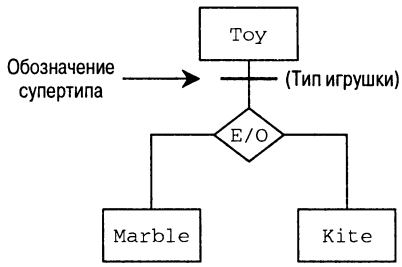


Рис. 5.13. Наследование в системе обозначений Чена для ER-модели

Относительно ER-моделей можно сделать два очень важных замечания. Поскольку не только объекты, но и сами связи в ER-моделях могут иметь атрибуты, связи представляют концептуальные объекты. Второе замечание заключается в том, что в ER-моделях Чена создание отношений в третьей нормальной форме (3НФ) рассматривается как нечто само собой разумеющееся.

Существует ряд методов разработки систем, в которых подсистемы моделирования данных в значительной степени базируются на реляционной модели. К ним относятся IE и SSADM, CASE*METHOD. Известно также несколько расширений к самой реляционной модели. Примечательным расширением этого типа является расширение RM/T Кодда, которое включает в реляционную алгебру неопределенные (null) значения и добавляет некоторые абстракции из ER-моделей, включая подтипы и множественное наследование, правила целостности, поддержку ограничений кратности и модальности, операции над базой данных и два различных типа отношений. E-отношения представляют существование сущностей, а P-отношения — их свойства. В этом направлении развивается бо́льшая часть реляционных СУБД (РСУБД); например, даже MS Access предоставляет возможность автоматического контроля целостности ссылок.

Известны два основных подхода к семантическим моделям данных. Первый подход основан на использовании функций или атрибутов для связи объектов, а второй — на применении конструкторов типа. Первый подход представлен в функциональной модели данных (Functional Data Model), а второй — в различных ER-моделях и их расширениях. Ключевое отличие между ними заключается в том, что в подходе с использованием конструкторов типов сами связи являются типами или объектами.

Функциональная модель данных [706] основана на лямбда-исчислении, которое кратко описано в главе 3. Простейшими понятиями в нем являются сущности, "функциональные" отношения и их так называемые инверсии. Функциональные отношения — это однозначные отношения или отношения "многие к одному". Наследование реализуется с помощью композиции функций. Одним из преимуществ логически обоснованной функциональной модели является то, что для нее существует формализованная теория. Как уже было упомянуто, существует ряд таких экспериментальных систем, как Daplex и Adaplex. Кроме того, на основе функциональной модели можно привести формальное обоснование для баз данных CODASYL или сетевых баз данных. Система Daplex, согласно [341], пригодна для представления объектов и богата вычислительными возможностями, но она не является полной. Для обеспечения полноты системы требуются перманентные языки программирования и объектно-ориентированные системы управления базами данных.

Общая семантическая модель GSM (Generic Semantic Model) [399] является попыткой создания супермножества на основе большинства семантических моделей данных, хотя сначала она была разработана в целях обучения. SDM — это другая модель, предложенная в [356] и [357]. В модели SDM, база данных — это непересекающиеся множества классов и экземпляров, в которых классы различаются по именам и значениям. Значение может быть экземпляром некоторого класса или именем примитива. Эта модель поддерживает наследование и включает язык предикатов для определения связей и производных данных аналогично дедуктивным базам данных. Коммерческая реализация SDM, названная SIM, описана в [419].

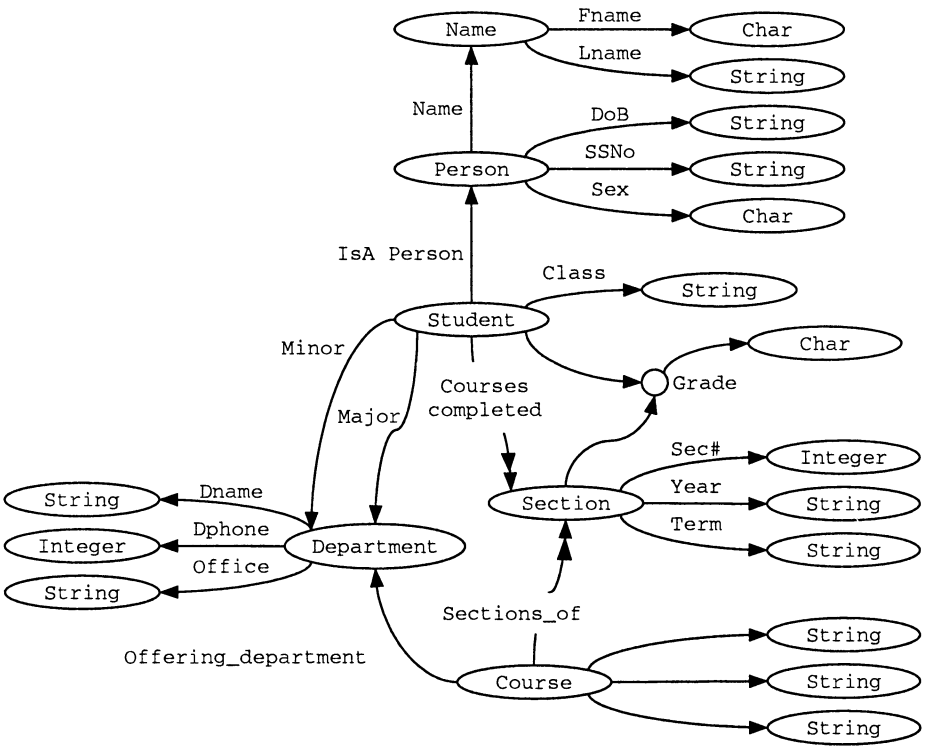


Рис. 5.14. Фрагмент модели данных для системы регистрации университетских курсов, представленной в типичной функциональной системе обозначений. Эллипсы являются объектами (множествами), а помеченные стрелки представляют функции (из [255])

Было предпринято несколько попыток сформировать реальные системы баз данных на основе полных семантических моделей. Система Adaplex является одной из таких ранних попыток реализации на верхнем уровне функциональной модели с помощью встраивания семантического языка базы данных. Система Daplex — это попытка реализации в рамках языка Ада. Система Distributed Data Manager [153] — это распределенная семантическая СУБД,

основанная на расширении системы Adaplex. Generis — это семантическая или "дедуктивная" база данных, основанная на расширенной двоичной реляционной модели. Существуют также базы данных типа "сущность-связь", основанные на модели Чена и некоторых специальных расширениях к реляционным системам управления базами данных, таких как Ingres/Postgres. Эти базы данных обладают возможностями как семантических моделей, так и объектно-ориентированных средств и систем искусственного интеллекта. Они рассматриваются ниже в этой же главе наряду с объектно-реляционными системами баз данных, такими как Oracle и Illustra (которая также известна как Informix Universal Server).

Объектно-ориентированные модели описывают скорее поведенческие абстракции (методы), чем структурные абстракции моделей данных. Они редко поддерживают конструкторы типов семантических моделей данных, например возможность моделировать функции, характеризуемые набором значений. Создание составных типов достигается посредством агрегирования атрибутов, как в случае создания типа `NAME` из атрибутов `STYLE`, `FORENAME` и `FAMILY_NAME`, или группирования их во множества, как в случае создания множества элементов подобного типа. Методы наследования, связанные с этими двумя подходами, также имеют свои отличия. Модели данных в случае систем искусственного интеллекта AI (Artificial Intelligence) часто обеспечивают наследование по умолчанию на уровне класса, в то время как объектно-ориентированные модели позволяют наследовать только имена атрибутов. В моделях данных наследование строго ограничено подтипами, в то время как в объектно-ориентированном программировании, помимо наследования данных (самих типов), возможно наследование методов. Например, текстовые объекты могут унаследовать метод "самоотображения", присущий геометрическим объектам `Geometric_Object`, хотя, по мнению автора, это — плохая практика.

В [450] подчеркиваются различия между семантическими моделями данных и объектно-ориентированными моделями.

> Семантические модели фокусируются на рекурсивном определении комплексных объектов, а также на наследовании структурных компонентов (агрегировании) и связях. Объектно-ориентированные модели фокусируются на определении и наследовании поведенческих возможностей в виде операций, встроенных в типы. Они поддерживают более простые возможности для структурирования комплексных объектов, но это различие не столь значительно, как кажется.

Однако, по мнению автора, существует необходимость в объектно-ориентированных базах данных и объектно-ориентированных методах вообще, призванных объединить в себе лучшие принципы структурного моделирования, сформулированные разработчиками моделей данных, с хорошими (по мнению автора) идеями исследователей и разработчиков систем искусственного интеллекта. В главе 6 автор попытается продемонстрировать, как это можно сделать на аналитическом уровне. Исследователи объектно-ориентированных баз данных тоже преследуют подобные цели, и системы ORION и Itasca, которые рассматриваются ниже в этой главе, представляют пример проектного решения с такими целевыми установками.

Система искусственного интеллекта, или, если строго, система, основанная на использовании знаний (knowledge-based system), по умолчанию устраняет конфликты наследования. Значительный вклад в теорию разработки систем вносит также программирование на основе правил. Этот метод будет более подробно описан в главе 6.

Для коммерческих приложений, таких как SACIS, где имеется важный компонент моделирования данных, полезно и даже необходимо разработать объектно-ориентированную систему моделирования, которая бы включала элементы семантических моделей данных и

объектно-ориентированного программирования. Подобное слияние или сближение объектно-ориентированного подхода с семантическими принципами моделирования данных и моделями искусственного интеллекта также, вероятно, будет полезно. Оно обеспечит разнообразие наследования, а также возможности образования групп и, в особенности, учета метаструктурных свойств этих групп, поскольку структура данных сама может быть свойством этих данных. С другой стороны, фреймовые модели искусственного интеллекта могли бы позаимствовать инкапсуляцию из объектно-ориентированного программирования, а также агрегирование и абстракцию из моделирования данных.

Покажем, почему реляционная модель не подходит для таких разработок.

5.2. Недостатки реляционной модели

Важным преимуществом реляционной модели является ее формальная обоснованность посредством логики предикатов первого порядка. Это свойство позволяет строить реляционно полные, непроцедурные языки запросов, такие как SQL или QUEL. Благодаря этой методологии некоторые свойства языков могут быть доказаны математически. Одним из примечательных и очень реальных преимуществ является возможность легкого изменения структуры базы данных без ее полного перепроектирования. Еще одно важное преимущество — это появление строгой теории нормализации.

Никто, возможно за исключением чрезмерно рьяных и алчных продавцов, не утверждает, что реляционная модель лишена недостатков. Недостатки, которые хочет рассмотреть автор, включают трудность выполнения рекурсивных запросов; проблемы, связанные с неопределенными (null) значениями; невозможность поддержки абстрактных типов данных; и, по мнению автора, наиболее важные, серьезные недостатки в представлении данных и функциональной семантике; отсутствие реальной поддержки бизнес-правил. Автор также хочет обратить внимание на некоторые проблемы, связанные с теорией нормализации. Ниже предлагается несколько более критическое рассмотрение теории нормализации, а также более глубокое рассмотрение других упомянутых проблем.

5.2.1. Нормализация

Проблемы с нормализацией совершенно очевидны для любого специалиста по объектно-ориентированным системам. Прежде всего, нормализованные отношения почти никогда не соответствуют какому-либо объекту в реальном мире. Разбиение на отношения производится скорее по принципам удобства вычисления или логики, чем в результате моделирования структуры приложения. Это не допускает автономного многократного использования нормализованных отношений, а также затрудняет сборку систем из компонентов. Нормализованные таблицы абсолютно ничему не соответствуют в приложении, и, к тому же, нормализация полностью скрывает семантику отношения. Причина в том, что нормализация маскирует сведения о приложении, которые представляются с помощью функциональных зависимостей между атрибутами. Это означает, что нормализация удаляет любую возможность полного восстановления структуры, и процесс нормализации является эффективно необратимым. Например, можно ли узнать из нормализованных отношений, представленных на рис. 5.15, что поставщики устанавливают цены в денежных единицах государства, в котором находится их город? Очевидно, нет, и потеря этой информации или функциональной зависимости помешает восстановлению структуры. Если денежные единицы изменятся (как это было сделано в пост–Маастрихтской Европе), то перепроектировать систему будет достаточно сложно.

Выше были рассмотрены некоторые из недостатков нормализованных моделей данных. Но не все, что касается нормализации, плохо. Устранение избыточности данных благодаря четвертой нормальной форме (4НФ) является неплохим показателем проектного решения, хотя эффективность поиска оставляет желать лучшего. Для объектно-ориентированной системы имеется аналогичная теория нормализации методов. По сути, инкапсуляция методов автоматически обеспечивает своего рода нормализацию. Например, в терминах сущностей и связей при рассмотрении связи `sold to` в системе SACIS видно, что эта связь должна гарантировать (посредством одного из ее методов) корректность выбора заказчика (`Customer`) и единицы товара (`item`). Было бы избыточным обеспечивать этот контроль целостности в классах `Customer` или `Stockltem`. При нормализованном поведении (normalized behaviour) эти данные наиболее естественно хранить в классе `SoldTo`.

Опытные разработчики моделей данных инстинктивно избегают в своих моделях связей "многие ко многим" (в качестве частичной нормализации). В результате этого вводятся сущности, которые часто не имеют соответствия в реальном мире. В объектно-ориентированном контексте это кажется совершенно неправильным. Обычным примером этого являются сущности `ORDERS` и `ITEMS`, которые связаны отношением "многие ко многим". Новая сущность, названная `ORDER-LINE`, используется для разрешения этой связи на две связи "один ко многим", как показано на рис. 5.15.

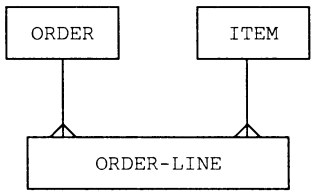


Рис. 5.15. Разрешение связей "многие ко многим"

Эта стратегия приемлема, когда имеется реальное, физическое соответствие сущностей, например, напечатанной строке заказа, которую можно увидеть. Однако в других случаях это абсурдно. В частности, правовое положение применимо ко многим товарам и производство каждого товара может подчиняться многим положениям. Каким будет эквивалент сущности `ORDER-LINE`? Может быть, `PRODUCT-REGULATION-INCIDENCE`? Аналогично автомобили могут быть окрашены во многие цвета, и одни и те же цвета могут применяться для различных моделей. Каким будет здесь эквивалент `ORDER-LINE`? В объектно-ориентированном контексте следует тщательно избегать присутствующей в моделях чрезмерной нормализации.

Тем, кого все еще волнуют проблемы, касающиеся аномалий модификации, следует обратить внимание на то, что эти аномалии возникают вследствие отсутствия объектной идентичности в реляционных системах, где таблицы и кортежи могут не соответствовать единственному объекту.

5.2.2. Правила целостности и бизнес-правила

Правила целостности — это семантические правила, утверждающие, что некорректно, например, удалять ссылку на поставщика лишь по той причине, что он не поставляет в данный момент времени ни одного вида товара. В реляционной терминологии можно устанавливать ограничения целостности сущностей и ссылок.

Целостность сущностей (entity integrity). Первичный ключ не может включать атрибут, который может принимать неопределенное (null) значение.

Целостность ссылок (referential integrity). Если отношение R содержит внешний ключ, значения которого соответствуют значениям первичного ключа отношения S, то каждое вхождение внешнего ключа в R должно либо соответствовать значению первичного ключа в R, либо быть неопределенным (null). R и S могут быть одним и тем же отношением.

Следует обратить внимание на то, что целостность сущностей не равносильна объектной идентичности, поскольку кортеж идентифицируется значением своего ключа, а не неизменной идентичностью, независимой от значений атрибутов.

Бизнес-правила — это утверждения вида "если служащий проработал более пяти лет, то ему предоставляется дополнительный день к отпуску" или, что менее жестко, "постоянные заказчики, которые регулярно оплачивают услуги, заслуживают специального внимания". Бизнес-правила устанавливают отношения между атрибутами сущностей.

В большинстве реляционных систем правила целостности часто кодируются в каждом клиентском приложении, что очень затрудняет внесение изменений. Более современной тенденцией является обеспечение возможностей преодоления этой трудности непосредственно на сервере баз данных. При реализации баз данных на платформе "клиент/сервер" также зачастую обеспечивается хранение на сервере "триггеров" — процедур, которые автоматически модифицируют таблицы при изменении значений некоторых данных. Этим достигается частичная поддержка бизнес-правил, но все же чаще всего они пишутся на языке SQL или на процедурном расширении SQL, что затрудняет их понимание и изменение. Поддержка правил любого вида является дополнением к реляционной модели, а не непосредственно ее частью.

5.2.3. Неопределенные значения

В кортежах неопределенные значения представляют трудность для интерпретации. Неопределенное значение можно считать логическим значением "неизвестно". Это может означать, что значение неприменимо к конкретным кортежам или что значения еще не были присвоены. В работе [215] предлагается их полностью запретить, а взамен ввести сложную суперструктуру новых логических операторов и значений по умолчанию. Однако неопределенное значение — это естественный способ оперировать неполными данными или исключениями. Трудность возникает из-за того, что FOPC является двузначной или булевой логикой, в которой допустимы только значения "истина" или "ложь". Разрешение неопределенных значений означает работу с логикой, допускающей три, четыре или даже бесконечное количество значений. Таким образом, наряду с очевидными выгодами работы в рамках формальной теории, этот подход накладывает ограничения, неприемлемые для решения реальных проблем.

Как и в случае с правилами, проблема неопределенных значений относится к семантике приложения. Реляционная модель является в чистом виде синтаксической моделью; она не содержит никакой внутренней поддержки для какой бы то ни было семантики. Однако необходимо учитывать семантику данных. Теме семантического моделирования данных посвящен огромный список литературы. Функциональная семантика также важна. Правила коммерции составляют часть ее функциональной семантики.

5.2.4. Абстрактные типы данных и сложные объекты

Первая нормальная форма запрещает хранение сложных объектов, так что реляционная модель не может оперировать абстрактными типами данных. Это означает, что запросы к базам данных, в которых хранятся неструктурированные большие двоичные объекты BLOB (Binary Large Object), содержащие структурированный текст, технические чертежи или графику, могут выполняться до невозможности медленно, поскольку вся структура найденного объекта должна быть восстановлена в приложении во время выполнения. Это также означает, что база данных не может хранить готовые для использования методы.

В реляционных или сетевых базах данных могут храниться связи, но они не являются динамическими, а создаются во время разработки базы данных. Кроме того, отсутствует естественный способ присваивания атрибутов связям — они не являются объектами. Атрибуты связей полезны для таких задач, как запись промежутка времени, в течение которого фиксируется связь. В частности, связь `supplies` (поставляет) может существовать только в определенное время года. Например, рождественские крекеры обычно имеются в наличии только в период с сентября по декабрь, хотя это и печально. Но, кажется, этот период из года в год становится все более длительным.

5.2.5. Рекурсивные запросы

Первая нормальная форма запрещает также рекурсивные запросы, т.е. запросы о связях сущности с самой собой. Примером такой связи являются родительские отношения личности.

Рекурсивные запросы были темой изучения в лаборатории IBM в Гейдельберге в рамках десятилетнего исследовательского проекта по расширенному управлению информацией (Advanced Information Management — AIM). Согласно результатам исследований [205, 488], короткий рекурсивный запрос вида

```
SELECT
FROM x IN Employees
WHERE x.Emp_ID = 'W09867'
```

на языке SQL может занять целых 18 строк. В проекте исследовалось расширение реляционной модели за счет нормализации отношений, но без использования первой нормальной формы. Разработчики назвали это расширение базой данных NF^2. Ненормализованные реляционные модели были использованы для автоматизации документооборота [452]. Следует обратить внимание на то, что определение нормальной формы Бойса-Кодда не зависит от наличия первой и второй нормальных форм, так что теория нормализации для таких баз данных все же допустима.

Единственный способ реализации рекурсивных запросов — это их реализация в виде поиска в иерархии наследования. В качестве примера можно рассмотреть следующий запрос в СУБД Adabas Entire, которая будет обсуждаться в следующем разделе.

```
Find person
 Referencing recursively
  via Child
   person = 'George'
```

Этот запрос находит всех предков Джорджа. Другими словами, отыскивается иерархия наследования вверх или в обратном направлении. Поиск вперед и вниз выполняется в соответствии с синтаксисом `Referenced recursively`.

Следует отметить, что для реализации этого запроса необходимо только одно отношение. Все типы хранятся в отношении `PERSON`, где явно указывается связь `child of`. Она хранится явно, поскольку в СУБД Adabas Entire реализована база данных типа "сущность-связь".

Стоит также отметить, что еще одним следствием первой нормальной формы и, следовательно, структур плоских файлов является необходимость объединения внешних ключей для реконструкции сложных сущностей в приложении. Это может оказаться очень ресурсоемкой (по времени) процедурой.

5.3. Базы данных типа "сущность-связь" и дедуктивные базы данных

Кроме обсуждаемых ниже полностью объектно-ориентированных баз данных, имеется несколько программных продуктов, в которых учтены все или некоторые из рассмотренных проблем. Их можно в общих чертах классифицировать как семантические базы данных. Коммерческие семантические базы данных можно разделить на четыре основных вида: базы данных типа "сущность-связь", дедуктивные базы данных, расширенные реляционные или объектно-реляционные базы данных и функциональные базы данных. Ниже рассматриваются примеры первых трех.

5.3.1. Базы данных типа "сущность-связь"

В СУБД Adabas Entire реализована база данных типа "сущность-связь", где связи между объектами сохраняются явно и, следовательно, могут быть восстановлены. Фактически СУБД Adabas никогда не настаивала на описании отношений в 1НФ, несмотря на утверждение, что она является реляционной базой данных. СУБД Adabas можно использовать как реляционную базу данных, если все отношения приведены к 1НФ. При этом бремя выбора правильного проектного решения возлагается на плечи разработчика базы данных. Что же касается баз данных вида NF^2, то их еще сложнее проектировать, как и все более мощные системы. Для их разработки требуется намного больше знаний и понимания. Тем не менее решение сложных проблем зачастую гораздо проще выразить в виде NF^2.

СУБД Adabas Entire непосредственно базируется на ER-модели Чена. Ее дополнительные функциональные возможности достигаются незначительным расширением синтаксиса Natural — языка Adabas четвертого поколения, который используется также и в качестве языка запросов. Явное сохранение связей не только допускает рекурсивные запросы, но и автоматически обеспечивает сохранение целостности ссылок. Как правило, в базах данных типа "сущность-связь", таких как Entire, нет понятий классов и наследования, в них также не предлагаются дополнительные возможности для хранения бизнес-правил (явно или на сервере).

Для преодоления этого недостатка фирма Software AG также разработала расширение Natural Expert, которое продавали как средство, позволяющее включать экспертные системы в приложения Natural. Однако основной отличительной особенностью этого расширения стало введение функционального программирования (фактически, формы метаязыка ML) вместо

более обычных продукционных правил. Функциональное программирование в объединении с базой данных типа "сущность-связь" позволило устранить довольно многие из упомянутых выше недостатков реляционных баз данных.

Сближение ER-модели и функциональных концепций, наблюдаемое в СУБД Adabas, становится очевидной тенденцией и при разработке функциональных баз данных, где сторонники систем типа Daplex рассматривают многие из своих работ как объектно-ориентированные. Иллюстрация этого сближения идей и стилей на примере конкретного проекта представлена в [340].

СУБД Adabas Entire никоим образом не является единственным в этой области программным продуктом. Одной из первых систем управления базами данных на основе модели "сущность-связь" является СУБД ZIM фирмы Zanthe.

5.3.2. Дедуктивные базы данных

Наряду с разработкой систем баз данных на основе модели "сущность-связь" и главным образом в результате достижений в области искусственного интеллекта и экспертных систем, была начата разработка ряда систем, которые можно охарактеризовать как "интеллектуальные" или дедуктивные базы данных. Многие из этих систем обладают всеми или некоторыми возможностями объектно-ориентированных баз данных наряду с возможностями экспертных систем. В этом разделе в качестве представителя этого вида систем рассматривается одна из мощных дедуктивных баз данных Generis.

СУБД Generis (первоначальное название Fact) является дедуктивной системой баз данных, базирующейся на расширении двоичной реляционной семантической модели, впервые представленной в [3], обобщенной реляционной модели (GRM) [539] и на традициях искусственного интеллекта. Эта система поддерживает явное хранение фактов (`A is X`), правил (`If P then Q`), связей, триггеров и сложных структур данных, а также необработанных (сырых) данных. Она поддерживает классы и множественное наследование с помощью формального описания фактов вида `A is B`. Эта система не содержала в себе никаких значительных возможностей для инкапсуляции, хотя один "метод" мог быть скрыт внутри определенных типов объектов. Чрезвычайно мощный язык запросов IQL, который позволяет вносить отсутствующую информацию в запрос, используя факты, хранящиеся в базе данных, делает программирование очень продуктивным. Для иллюстрации приведем запрос на языке SQL.

```
SELECT      EMPLOYEE, SALARY, DEPARTMENT
FROM        EMPLOYMENT, DEPARTMENTS
WHERE       LOCATIONS='LONDON'
AND         JOB_TITLE='SALESMAN'
AND
EMPLOYMENT.DEPARTMENT=DEPARTMENTS.DEPARTMENT
```

На языке IQL он будет выглядеть так.

```
DISPLAY SALARY SALESMAN AND DEPARTMENT FOR LONDON
```

Результаты этих запросов совпадают, но в Generis связи (такие, как `works in DEPARTMENT of employee`) отыскиваются и выводятся в виде заголовков столбцов, как показано на рис. 5.16.

Другими словами, структура класса, или знание, содержащееся внутри него, используется для заполнения отсутствующих связей. СУБД Generis также поддерживает рекурсивные запросы и обладает возможностями, которые сделали ее особенно удобной для интеллектуальных приложений поиска текстовой информации. Она использовалась в ряде таких приложений, особенно в области здравоохранения. СУБД Generis является необычным и мощным гибридом, основанным на идеях реляционных, объектно-ориентированных и экспертных систем. Это расширило реляционную модель за счет включения семантики данных и обеспечило возможность использования рекурсивных запросов и наследования, но не абстрактных типов данных или полную инкапсуляцию. С профессиональной точки зрения одним из наиболее интересных потенциальных приложений таких семантически богатых языков является создание прототипа спецификации. К сожалению, в настоящее время имеется немного таких коммерчески доступных программных продуктов.

EMPLOYEE	works in DEPARTMENT of employee	located in LOCATION of department	Earns SALARY of employee
A. Langham	Sales	London	15000
R. Biter	Production	London	27000

Рис. 5.16. Стандартный формат отчета в СУБД Generis

Известно несколько иных подходов к созданию дедуктивных баз данных. Один из них, причем достаточно популярный, состоит в добавлении возможностей логического программирования к существующей РСУБД с использованием в качестве расширения базы данных языка, подобного языку Prolog. Исследованию этого подхода посвящена работа [69]. Другие дедуктивные базы данных рассматриваются в [341]. При этом их подходы сравниваются с принципами объектно-ориентированного программирования.

5.4. Объектно-реляционные базы данных

До настоящего момента в этой книге были рассмотрены преимущества объектно-ориентированных языков программирования и некоторые из ограничений традиционных систем управления базами данных, для которых были предложены пути преодоления. Однако в объектно-ориентированных языках программирования также имеются свои ограничения. В частности, в языках, подобных Simula, C++ и Smalltalk, отсутствует явная поддержка файловых (или перманентных) объектов, т.е. объектов, которые автономно хранятся на вспомогательных запоминающих устройствах и состояние которых от момента к моменту и из сеанса в сеанс сохраняется. Такие языки не имеют эффективных средств для оперирования перманентными объектами по принципу баз данных. Нельзя отправить сообщение объекту, хранящемуся автономно. Например, самое большее, что можно реализовать в Smalltalk, это автономное хранение образа сеанса и его загрузку в следующем сеансе. Очевидно, это не отвечает требованиям приложения, где необходимо оперировать большим количеством важных данных, которые необходимо сохранять. А от коммерческих приложений требуется именно эта возможность.

В Java, который не является перманентным языком (persistent language), классы позволяют реализовывать стандартный интерфейс `serializable`, обеспечивающий в некоторой степени эту возможность. Но даже полностью перманентные языки не помогают в решении таких проблем, как взаимная совместимость, безопасность и восстановление данных. Для решения этих проблем в таких языках программисту все же приходится писать дополнительный код.

Вопрос заключается в том, каким образом можно использовать преимущества объектно-ориентированного подхода к разработке систем и при этом сохранить способность автономного хранения файловых объектов и управления ими, а также как справляться с такими проблемами, как взаимная совместимость, безопасность и восстановление данных, столь же естественно, как в реляционных системах управления базами данных. Несколько подходов к решению этой проблемы появилось под лозунгом объектно-ориентированных баз данных.

Можно назвать три причины необходимости объектно-ориентированных баз данных. Это реализация ясного интерфейса с объектно-ориентированным языком программирования, поиск решений для приложений, которые требуют гибкости реляционной базы данных и более высокой производительности, а также поиск абсолютно новых видов приложений, для которых модельное представление передачи сообщений кажется наиболее подходящим. Примечательно, что чисто объектно-ориентированные базы данных в корректно написанных приложениях могут сохранять гибкость реляционной реализации в терминах эволюции схемы и при этом работать более чем в 100 раз быстрее.

Коммерческая потребность в полностью интегрированных объектно-ориентированных системах привела к разработке ряда систем, характеризуемых как объектно-ориентированные системы управления базами данных. В свое время были главным образом экспериментальные системы, но такие программные продукты, как GemStone, Jasmine, ObjectStore, POET и Versant, теперь появились в качестве вполне законченных коммерческих продуктов. Другие же, подобно Ontos, как возникли, так и исчезли. Также имеется ряд новых систем баз данных с объектно-ориентированными возможностями, включая дедуктивные и семантические базы данных, которые не являются в полном смысле объектно-ориентированными системами баз данных, но используют многие из их возможностей и преимуществ. Развитие объектно-ориентированного подхода привело даже к кратковременному возрождению интереса к иерархическим и сетевым базам данных. База данных типа "сущность-связь" Adabas Entire, в конечном счете, была реализована как сетевая база данных. В [765, 766] обращается внимание на то, что иерархические и сетевые базы данных, такие как IMS и базы данных семейства CODASYL, поддерживают объектную идентичность, хотя и не обладают другими возможностями объектно-ориентированной модели. Их основное ограничение — негибкость в отношении эволюции схемы. Оно является следствием способа реализации идентичности объектов в виде фиксированных указателей.

Ключевой вопрос для разработчиков стратегии системы состоит в следующем: нужно ли переходить на новые объектно-ориентированные базы данных, ничего не делать или предпочесть стратегию, базирующуюся на расширениях реляционных продуктов. Автор попытается обосновать решение этой проблемы.

Как показано на рис. 5.17, в мире господствуют четыре типа перспективных программных продуктов.

	Механизм реляционной базы данных	Механизм объектной базы данных
Реляционная модель данных	**SQL Server, Sybase**	**Cincom ORDB**
Объектная модель данных	**DB2, Illustra, Oracle**	**GemStone, Itasca Jasmine, ObjectStore, POET, Versant**

Рис. 5.17. Типы программных продуктов объектных баз данных

Безусловно, система Cincom ORDB (прежнее название UniSQL) принадлежит к в высшей степени уникальной категории: в ней сочетаются гибкие возможности реляционных запросов с эффективностью и богатством семантики объектно-ориентированного подхода. Этот продукт не будет рассматриваться подробно. Отметим лишь, что он является интеллектуальным потомком объектно-ориентированной базы данных Orion, которая будет рассматриваться ниже. Теперь перейдем к обсуждению объектно-реляционных баз данных.

Ряд производителей реляционных систем управления базами данных (РСУБД) в результате достижений в области объектно-ориентированного программирования и экспертных систем добавили значительные семантические расширения к своим программным продуктам. Эти объектно-реляционные базы данных теперь конкурируют за место на рынке с подлинными объектно-ориентированными базами данных и заменяют РСУБД для многих приложений.

Реляционная СУБД Sybase была первой системой, которая продемонстрировала новаторские возможности этого вида. Она базируется на архитектуре клиент/сервер, в силу чего ограничения целостности данных и бизнес-правила могут храниться на сервере в базе данных, а не в каждом индивидуальном приложении. Это облегчает сопровождение системы, потому что при изменении правила не нужно проверять каждое приложение. Правила реализованы в виде хранимых процедур и триггеров — активных или управляемых данными правил, которые запускаются при изменении данных. Трудность состоит в том, что эти правила реализуются на SQL или процедурном языке и не обязательно просты для понимания или изменения. При этом они не могут быть сохранены на отдельном уровне. Эта проблема решается с помощью нестандартного процедурного расширения языка SQL, Transact SQL, которое включает управляющие структуры, процедуры, операторы присваивания и другие универсальные программные конструкции. Таким образом, Sybase предлагает в некоторой степени скромные, но важные расширения унифицированной реляционной системы. Добавление хранимых процедур к таблицам базы данных является, несомненно, шагом, приближающим систему к объектно-ориентированной парадигме, которой свойственно объединение данных и процессов. Однако истинная инкапсуляция здесь не реализована, так как к данным можно обращаться непосредственно и таблицы не соответствуют объектам предметной области вследствие нормализации. Следовательно, Sybase не является полностью объектно-реляционной базой данных.

СУБД Ingres, начиная с версии 6, предлагает подобные расширения в виде многоклиентской/многосерверной архитектуры, что очень важно с точки зрения производительности и создания распределенных баз данных, а также возможности поддержки правил и триггеров. Однако СУБД Ingres пошла дальше Sybase и других реляционных систем в обеспечении явной поддержки для абстрактных типов данных, правил, рекурсивных запросов и наследования. Любопытно, что СУБД Sybase и СУБД Ingres имеют общие исторические корни в том смысле, что некоторые разработчики Ingres после некоторого периода работы над аппаратными средствами базы данных создали Sybase.

Примечательно, что идеология Ingres минимально расходится с реляционной моделью. Это сделано во имя спокойствия компаний, обладающих большими ресурсами и преданных реляционному подходу. В ней в виде отдельного уровня реализован диспетчер объектов, но Ingres все же можно использовать как чисто реляционную базу данных. В Ingres также используется диспетчер знаний, реализованный в виде отдельного уровня по тем же соображениям.

Мнение о том, что объектная ориентация в базах данных и коммерческих системах является обязательным признаком развития программных средств, связано с ранним появлением стандартов, представленных влиятельными участниками рынка: с одной стороны, группу ODMG (Object Database Management Group — рабочую группу по выработке и согласованию стандартов объектных баз данных, включающую ведущих разработчиков объектных СУБД), а с другой — комитет по усовершенствованным функциям СУБД CADF (Committee for Advanced DBMS Function).

Комитет CADF заложил теоретическую основу для объектно-реляционных баз данных и поддерживаемого ими языка запросов SQL3. В этот комитет вошла группа влиятельных личностей из мира баз данных, которые в 1990 году опубликовали своего рода манифест усовершенствованных баз данных [731]. Первоначально комитет CADF состоял из таких фигур, как Майк Стоунбрейкер (Mike Stonebraker) и Лэрри Роу (Larry Rowe) — главные архитекторы СУБД Ingres, Дэвид Бич (David Beech) — технический консультант Oracle, один из разработчиков IRIS, ранее вовлеченный в определение языка SQL3, Джим Грей (Jim Gray) — автор языка Nonstop SQL для СУБД Tandem, Фил Бернстайн (Phil Bernstein) — руководитель лаборатории Digital DBMS, Брюс Линдсет (Bruce Lindset) — архитектор DB2 и Майкл Броди (Michael Brodie) из GTE — ведущий исследователь в области систем управления базами знаний. Группа установила ряд принципов, которым в ближайшем будущем должна отвечать усовершенствованная система управления базами данных. Эти принципы, аналогичные знаменитым 12 правилам Кодда для реляционных систем, включают следующее.

- Объектно-ориентированные базы данных должны обладать всеми возможностями традиционной СУБД, включая обработку большого объема данных, параллельный многопользовательский доступ, высокоуровневые языки доступа, одновременный доступ к распределенным данным, поддержку эффективной обработки транзакций и возможности полного восстановления и безопасности.
- Должна поддерживаться объектная идентичность.
- Должна поддерживаться, по меньшей мере, одна форма наследования.
- Необходимо обеспечить прозрачный распределенный доступ и возможность модификации.
- Необходима поддержка сложных объектов через инкапсуляцию или абстрактные типы данных и классы.
- Обеспечение богатых в вычислительном отношении языков, сравнимых с SQL.
- Поддержка динамического связывания.
- Поддержка определяемых пользователем функций.
- Контроль соответствия типов и возможности логического вывода.
- Эффективное управление версиями и конфигурацией.

Система Illustra (на сегодняшний день объединенная с Informix и переименованная в Informix Universal Server — чересчур многословное название, чтобы использовать его многократно) является промышленным объектно-реляционным преемником СУБД Ingres и Postgres. Illustra поддерживает все перечисленные выше объектно-ориентированные возможности на верхнем уровне реляционного механизма хранения и в качестве языка запросов использует SQL3. Как и СУБД Ingres, Illustra имеет многоклиентскую/многосерверную архитектуру и прекрасный оптимизатор запросов. Методы (в отличие от некоторых чисто объектно-ориентированных СУБД (ООСУБД)) могут храниться в базе данных. Определяемые пользователем типы называются DataBlades и обеспечивают оптимизированный внутренний доступ к временным последовательностям, изображениям, рисункам или Web-страницам. Другие объектно-ориентированные возможности этой системы включают множественное наследование, полиморфизм и инкапсуляцию. Каждая запись в базе данных также содержит идентификатор объекта OID (Object IDentifier), обеспечивающий использование навигационного доступа. Однако в [748] показано, что это подрывает математические основы реляционной технологии. Идентификатор объекта OID в Illustra характеризуется значением, а не ссылкой, и является просто потенциальным ключом (candidate key).

Корпорация Oracle также начала добавлять объектно-ориентированные возможности к своему программному продукту, который, начиная с версии 8, превратился в почти полностью объектно-реляционную базу данных (там также используется язык SQL3). Oracle и Illustra были оптимизированы для более быстрого выполнения объединения, чем предусматривали их предшественники, с использованием, главным образом, метода усовершенствованного страничного буфера. Заметным различием между двумя программными продуктами является то, что Oracle, в отличие от Illustra, не имеет никакой встроенной поддержки продукционных правил, свойственных экспертным системам. Система DB2 — это универсальная база данных IBM, которая также поддерживает возможности, провозглашенные манифестом комитета CADF.

Группа ODMG также подписалась под манифестом объектно-ориентированных баз данных [44] и заложила основу для стандартов чисто объектно-ориентированных баз данных и соответствующего языка запросов OQL. Ключевым различием между чисто объектно-ориентированными базами данных и реляционными гибридами, которые были упомянуты выше, все еще является производительность, хотя, как будет видно из дальнейшего, она зависит и от типа приложения.

Спектр программных продуктов компании Persistence Software обеспечивает компромисс между объектно-ориентированными языками программирования и реляционными базами данных: объекты языка неявно отображаются в таблицы баз данных, а для повышения производительности обеспечивается кэширование. Здесь реализованы собственные принципы кэширования, отличные от традиционных приемов, принятых в коммерческих объектно-ориентированных базах данных для кэширования объектов, хранящихся в реляционных базах данных. Перманентность (способность программного обеспечения создавать и поддерживать перманентные объекты (persistent object)) согласуется с технологией CORBA (Common Object Request Broker Architecture), представляющей собой предложенную фирмой IBM технологию построения распределенных объектных приложений. Она использовалась при разработке серьезных приложений такими компаниями, как JP Morgan и Fedex.

В системе Cache от InterSystems предложен несколько иной подход к идее гибридных объектно-реляционных баз данных. Данная система базируется на доменно-реляционном или многомерном представлении, подобно рассмотренной выше системе Oracle Express. Отображать объекты в многомерное представление намного легче, чем приводить их к

чисто реляционному виду; следовательно, производительность такой системы должна быть выше. Однако все имеет свою цену. Система Cache специально предназначена для приложений с высокой интенсивностью обработки транзакций в интерактивном режиме.

5.5. Языки запросов

Обычно коммерческие системы оперируют большими объемами данных, но сами операции относительно примитивны. По этой причине выразительная мощность большинства языков обработки данных очень ограничена. Сложные процедуры реализуют на таких универсальных языках, как COBOL, C или FORTRAN. Базы данных, которые должны иметь дело с более сложными связями, в том числе GIS, проект VLSI или CAD, требуют намного большей общности языка доступа. Этим объясняется необходимость разработки богатых в вычислительном отношении языков, т.е. языков, которые предоставляют возможность лаконичного выражения любых процедурных или непроцедурных вычислений.

Абстракция в базах данных означает необходимость извлечения меньшего объема данных, поскольку сервер лучше "понимает" семантику объекта, а не просто извлекает недифференцированную строку битов (BLOB), а затем восстанавливает всю структуру. При этом задача извлечения соответствующих данных возлагается на приложение, обладающее семантическими знаниями.

Без стандартов технологию всерьез не примет ни одна серьезная коммерческая организация, а если и примет, то только в виде исключения для некритичных приложений. Выпуски ключевых стандартов для объектно-ориентированных баз данных включают точное определение объектной модели, описание расширений для языка SQL и даже описание стандартных расширений обработки данных для языков, подобных C++, по меньшей мере на уровне потребителя. В настоящее время имеется два стандарта для языков запросов: язык SQL3, являющийся комплексным объектно-ориентированным расширением языка SQL92 и поддерживаемый производителями объектно-реляционных баз данных, и OQL — более простой язык, поддерживаемый производителями чисто объектно-ориентированных баз данных и основанный на объектной модели рабочей группы по развитию стандартов объектного программирования OMG (Object Management Group).

Цель языка SQL3 состоит в том, чтобы облегчить эволюцию схемы. В нем вводится понятие типа, отличного от таблиц. Тип кортежа может не совпадать с типом содержащей его таблицы. Разрешается использование подтипов, т.е. создание новых таблиц данного типа на основе следующего синтаксиса.

```
create object type PartTimers under Employees
```

В таблицах, не удовлетворяющих первой нормальной форме, разрешается использование атрибутов со значениями кортежей. Идентичность объектов поддерживается с помощью ссылочных REF-типов, которые дают возможность типизации атрибутов, а не средств системы, генерирующих недоступные для редактирования смешанные и внешние ключи. Наконец, типы могут содержать определения функций или методов, которые могут быть описаны процедурными средствами языка SQL3. Атрибуты, как и функции, могут быть закрытыми или общедоступными. В [67] отмечено, что язык SQL3 устраняет все конфликты между реляционной моделью и инкапсуляцией.

В языке SQL3 также поддерживаются абстрактные типы данных, триггеры, рекурсивные запросы, квантификация, а применение команд `Grant` и `Revoke` повышает безопасность. Это полноценный язык программирования, реализующий машину Тьюринга. Язык OQL является частью стандарта ODMG-93 [150]. Этот стандарт включает также язык описаний объектов, базирующийся на языке описания интерфейсов IDL (Interface Definition Language) группы OMG и связанный с такими объектно-ориентированными языками программирования, как C++ и Java. Синтаксис языка OQL напоминает синтаксис SQL92 (`select/from/where`). В отличие от языка SQL3, OQL не является полноценным языком программирования, следовательно, он проще для изучения.

Большинство промышленных ООБД не только поддерживают язык OQL, но и тесно связаны с конкретным объектно-ориентированным языком программирования. Однако в них обычно предлагается программный API-интерфейс и для других языков. Это может усложнить интеграцию существующего кода с базой данных. Теперь перейдем к рассмотрению чисто объектно-ориентированных систем управления базами данных.

5.6. Что такое объектно-ориентированная база данных

В главе 1 было сказано, что объектно-ориентированный язык программирования обладает тремя основными свойствами.

- Инкапсуляция — возможность работать со сложными объектными интерфейсами и заключать в них методы и структуры данных
- Наследование — возможность группировки объектов и совместного использования атрибутов и методов
- Самоидентичность или объектная идентичность — свойство, на которое до сих пор обращалось мало внимания

Обычные объектно-ориентированные языки программирования не поддерживают некоторые основные функции, необходимые для приложения, в котором обрабатывается большое количество данных (data-intensive application). В частности, в таких языках не имеется никакой поддержки для:

- перманентных объектов;
- управления очень большими объемами данных;
- целостности данных и транзакций;
- параллельного многопользовательского доступа;
- языков доступа или запросов;
- восстановления;
- безопасности.

Файловые, или перманентные, объекты (persistent object) — это объекты, которые продолжают существовать после того, как сеанс завершается. Объекты могут создаваться и уничтожаться в памяти приложением или компилятором. Существует два способа обработки перманентных объектов. Эти объекты можно хранить в традиционной базе данных. Это означает, что объект не готов для использования; он должен быть построен на основе извлеченной из файлов информации. Если используется объектно-ориентированная база данных и объекты сохраняются непосредственно, то каждый такой объект сразу же готов для использования. Перманентность тесно связана с объектной идентичностью, потому что при связывании объектов через ссылки на уникальные идентификаторы связи между объектами тоже являются перманентными.

Способность системы идентифицировать объекты в течение времени их существования с помощью уникального идентификатора, а не их компонентов или атрибутов называется **объектной идентичностью**. Существует три стандартных метода идентификации объектов: с помощью их физического расположения в памяти (указателей), на основе первичных ключей или посредством идентификаторов, определяемых пользователем. В реляционной системе при изменении первичных ключей может также изменяться объектная (кортежная) идентичность. В таких системах идентичность должна имитироваться с помощью смешанных или уникальных идентификаторов, таких как Supplier_No. Но поскольку они не находятся под контролем системы, это не обеспечивает гарантию идентичности. К объектам можно обращаться различными путями, связываясь с различными переменными. При этом нельзя определить, относятся ли две переменные к одному и тому же объекту. Использование концепции первичных ключей также проблематично. Первичные ключи не могут изменяться. Если же ключ представляет свойство, которое может изменяться (а есть такие, которые не могут?), то это вызывает серьезные проблемы. Кроме того, если необходимо объединить две системы, которые используют различные ключи для одного и того же объекта, скажем `name` и `employee number` для служащих, то в результате изменения может нарушиться целостность идентичности. Обычно такие ключи являются уникальными только внутри отношений, но не для всей системы. Например, если Сэм является и служащим и акционером одной и той же компании, то он может храниться в виде двух сущностей, а не одной, что ведет к потенциальной противоречивости и избыточности. Поэтому не стоит удивляться, что одно и то же электронное напоминание приходит по нескольку раз.

Имеется несколько способов реализации объектной идентичности. Чаще всего используется сгенерированный системой уникальный идентификатор. С его помощью преодолеваются упомянутые выше проблемы, и он обычно применяется как для объектно-ориентированных баз данных, так и для объектно-ориентированных языков. Объектная идентичность при этом не будет зависеть ни от расположения объекта, ни от его свойств. В реальном мире большинство свойств объекта тоже может изменяться, но объект все же будет оставаться самим собой. Хорошим примером является река, которая с течением времени может менять свои истоки, направление и размеры, но все же останется Амазонкой.

Объектная идентичность объясняет и расширяет понятие указателей, используемых в иерархических базах данных и стандартных языках, подобных C. Для связи между объектами можно использовать хранимую идентичность, т.е. выражение, ориентированное на значение, как в языке SQL. Таким образом, подход объектно-ориентированных баз данных, в некотором смысле, унифицирует доступ по значению и идентичности.

Объектно-ориентированная система баз данных (или модель данных) обладает всеми возможностями баз данных вообще, включая языки доступа, способность управлять очень большими объемами данных, перманентность, а также обеспечение целостности данных и

транзакций, параллелизма, безопасности и восстановления. Кроме того, она поддерживает инкапсуляцию, наследование и объектную идентичность. Более того, современные объектно-ориентированные базы данных обычно обеспечивают некоторый контроль версий (versioning) системы. Последнее является в значительной степени исторической случайностью. В то время как реляционные базы данных, в основном, разрабатывались в эру изолированных миникомпьютеров или универсальных вычислительных машин, объектно-ориентированные базы данных появились в эпоху рабочих станций и сетей. Это означает, что новые программные продукты были разработаны в пределах культурной среды, где обмен результатами с коллегами по сети является совершенно нормальным. Транзакции этого вида называются **длинными транзакциями**. Они часто происходят в приложениях технического проектирования и изготовления документов, но не в приложениях, где обычно используется традиционное управление базой данных. Традиционные базы данных могут управлять только короткими транзакциями. Например, можно представить, что произойдет в случае конфликта транзакций двух пользователей банкомата ATM (Automatic Teller Machine), снимающих наличные деньги. Обычно, одна из транзакций прерывается, и пользователь ATM должен ретранслировать команду. Возможно, это является раздражающим фактором, но это не катастрофа. Теперь представим аналогичную ситуацию, когда два инженера, которые потратили на свои проекты месяцы работы, одновременно хотят передать их в базу данных. Можно только предположить, что почувствовали бы большинство инженеров, если бы их транзакции были прерваны и потеряны. Здесь требуется автоматическое создание временной версии транзакции и ее фиксация, а не просто аварийное окончание. Таким образом, контроль версий является необходимостью, и он был встроен в программные продукты объектно-ориентированных баз данных начиная с самых ранних проектов.

Объекты могут сохранять связи с другими объектами и методами. Эти связи часто рассматриваются как полноправные объекты, которые также могут иметь атрибуты и методы. Например, связь использования может быть зависимой от времени, поскольку использование офисного телефона автора может быть ограничено некоторыми временны́ми промежутками или зонами. В этом примере отношение использования может включать методы управления, касающиеся сообщений о международных телефонных звонках определенной части персонала. Таким образом, объектно-ориентированные базы данных содержат многие функциональные возможности семантических моделей данных, хотя и обладают некоторыми возможностями наследования, которые не всегда поддерживаются в объектной модели. Объектно-ориентированные базы данных идут дальше семантических моделей, поскольку они фиксируют поведенческие абстракции или методы. Это означает, что они могут представлять и даже нормализовать поведение.

В объектно-ориентированной базе данных вместе с объектом хранятся как атрибуты, которые сами могут быть сложными объектами, так и методы. При этом программный код комплексных объектов и методов не может быть физически сохранен с идентификатором объекта, но указатели на физическое расположение могут быть сохранены. Важно, что любой язык запросов справляется с этой проблемой физического хранения вполне прозрачно, и пользователь или проектировщик приложения базы данных может считать, что принципы физического хранения соответствуют концептуальной модели. В действительности только немногие из промышленных объектно-ориентированных баз данных хранят методы. Большая часть хранит только структуры данных; методами оперирует система поддержки выполнения программ традиционным способом. Возможно, это оправдывает использование термина "объектно-ориентированная база данных" вместо другого более предпочтительного, но никогда неиспользуемого "объектная база". Вследствие этого большинство программных

продуктов тесно связано с конкретными объектно-ориентированными языками программирования, как это будет видно в разделе 5.8.

В объектно-ориентированном программировании акцент делается на поведенческой абстракции. Это означает, что основное внимание уделяется наследованию инкапсулированных методов. В объектно-ориентированных системах баз данных выдвигается более жесткое требование моделирования структурных свойств данных, т.е. явное наследование и агрегирование структур. Этот вид иерархии можно создавать несколькими различными способами. Объекты могут быть сгруппированы в классы в соответствии с их общими структурными и поведенческими свойствами. Это называется **классификацией**. Например, служащие и заказчики могут быть сгруппированы в классе людей, а люди, животные и растения — в классе живых сущностей. Однако не существует никакого уникального способа классификации объектов, и реальные объекты могут быть классифицированы как принадлежащие к нескольким классам одновременно. Структуры наследования классов (или схемы) представляют обобщение и специализацию. Транспортные средства являются обобщением автомобилей и специализацией предметов, созданных руками человека. **Ассоциация** тесно связана с классификацией. Здесь конкретные экземпляры группируются в классы по некоторому общему свойству. Примером ассоциации объектов, отличным от их классификации, может служить множество предметов красного цвета или множество предметов весом более одной тонны. Различия этих двух способов группировки и их использование при анализе данных описываются в разделе 6.4.1, где рассматриваются существенные (классифицированные) и случайные (ассоциативные) объекты. Классификация иногда представляет отношение подмножества, а ассоциация — связь принадлежности. Некая сущность, которая является экземпляром в одном приложении, может быть классом в другом. Например, в базе данных научных терминов, `hominid` (гоминид) является экземпляром, в то время как в антропологической базе данных гоминиды могут иметь несколько экземпляров, т.е. являются подклассами. Объекты могут быть сгруппированы как компоненты некоторого составного объекта двумя способами: в качестве части структуры или множества атрибутов. Это соответственно называется **композицией** (composition) и **агрегированием** (aggregation). Например, автомобиль состоит из колес, коробки передач, корпуса и т.д. Это можно назвать "структурой композиции". Структура агрегирования — это более общее понятие, которое включает возможность концептуальной и физической композиции. Например, компьютер — это концепция, агрегированная из концепций, включающих имя, операционную систему, другое программное обеспечение, расположение, права доступа и т.д. Необходимо обратить внимание на то, что эти компоненты могут в свою очередь агрегировать, например, другое программное обеспечение. Агрегирование атрибутов и методов — это способ концептуального формирования объектов в объектно-ориентированном программировании и проектировании.

Важно обратить внимание на то, что наследование и схемы агрегирования могут взаимодействовать. Объект, который является агрегатом нескольких объектов, содержит (наследует) их атрибуты и методы. Объект, который наследует атрибуты и методы от супертипа (данных), наследует также их структуру агрегации. Эта проблема возникает в объектно-ориентированном программировании не часто, но в базах данных она является критической. Теория этого взаимодействия разработана еще не полностью, но теория семантических моделей данных позволяет решить многие проблемы.

В проектировании сложных систем обработки данных важную роль играет структура **использования** (usage) или взаимодействия между объектами на платформе "клиент/сервер". Эта структура представляет топологию передачи сообщений в системе и определяет сложность управления ею. Эти вопросы будут рассматриваться в главе 6.

Как будет ясно из дальнейшего, каждый глагол может давать начало структуре ассоциации. Особую роль играют уже упомянутые глаголы, а именно: использовать, агрегировать и быть видом (чего-то или кого-то). В системах искусственного интеллекта, где используются семантические сети, иногда встречаются другие структуры, базирующиеся на связях принадлежности (ownership). Попытки установить фиксированные совокупности "концептуальных примитивов" в целом потерпели неудачу. Авторы работы [683] пытались сделать это в рамках своей теории "сценарных" объектов. Ленат и Гуа [483] также потерпели неудачу со своей системой CYC. По мнению автора, это отражает аспекты общефилософской теории категорий. Аристотель определил совокупность фиксированных категорий мысли, которая была подхвачена и модифицирована Кантом. Потребность пересмотра этой совокупности приводит к наблюдению, что эти категории обусловлены социальными и историческими факторами и не могут быть фиксированными. В человеческой практике определяются канонические категории. В частности, в области обработки данных каноническими категориями или структурами являются композиция, классификация и использование. Семантические модели данных, которые уже рассматривались в этой главе, в том числе EER- и GSM-модели, являются достаточно выразительными, чтобы фиксировать семантику классификации, агрегирования, ассоциации и использования, но слабы в описании поведенческих показателей. Они обеспечивают базис для описания схемы в объектно-ориентированных базах данных. Потребность в комплексных, взаимодействующих структурах этого вида возникает потому, что они отражают структуру реального мира и являются частью семантики приложения базы данных. Однако раскрытие атрибутов и методов посредством этих структур затрудняет ввод в действие принципов сокрытия информации (от пользователя). Следовательно, в объектно-ориентированной базе данных требуется гораздо больше операций со структурами данных, чем в объектно-ориентированном языке программирования.

Чтобы уточнить это нестрогое определение, нужно провести параллель между объектно-ориентированными и реляционными базами данных. Как убедительно аргументирует Ульман [765, 766], существует компромисс между непроцедурностью (или недекларативностью) и объектной идентичностью. Реляционные системы предлагают декларативные языки доступа, базирующиеся на логике оптимизации запроса. Объектно-ориентированные системы по своей природе являются навигационными и недекларативными. Это обеспечивается явным объявлением связей между объектами и иерархиями классификации и агрегирования. Однако по той же причине такая система требует меньшей оптимизации, поэтому для запросов по сложным объектам она может оказаться более эффективной.

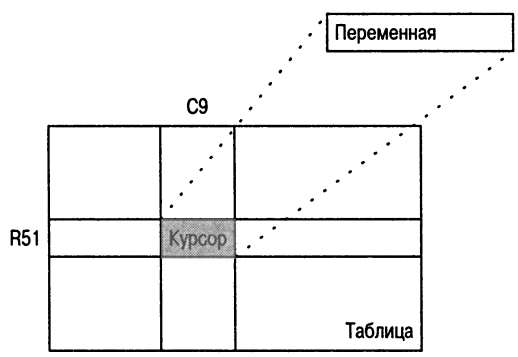


Рис. 5.18. Несоответствие импеданса

Несоответствие между ориентированными на работу с таблицами языками, такими как SQL, и обычными языками программирования, ориентированными на записи, означает, что для доступа к данным требуется отдельный язык управления данными DML (Data Manipulation Language). В реляционной системе единицей измерения является таблица; каждый запрос SQL возвращает таблицу. В стандартном языке программирования объект данных (единица обращения) — это переменная. Для решения этой проблемы программисты баз данных обычно пишут программный код для перемещения "курсора" по таблицам и преобразования ячеек (таблицы) в переменные и наоборот. Это **несоответствие импеданса**[3] (impedance mismatch) между декларативными и процедурными стилями (рис. 5.18), а также между различными типами систем, отличающихся языком приложений и баз данных, вызывает потерю информации при взаимодействии и препятствует автоматическому контролю соответствия типов. В языках четвертого поколения 4GL (4 Generation Language) эти проблемы частично преодолеваются. В объектно-ориентированных системах баз данных имеется соответствие между языками программирования и языками запросов, которые базируются на понятии объекта. Можно легко создать богатый в вычислительном отношении язык DML или расширить прикладной язык, чтобы включить в него систему типов баз данных. Тогда различие между двумя типами языков может почти исчезнуть. Например, объектно-ориентированная система баз данных Versant реализует все запросы на расширенном языке Java или на языке C++. Как будет видно ниже (глава 6), это также относится к различию между языками и методами; язык и модель определяют метод. Разрешение проблемы несоответствия импеданса связано с использованием распределенных архитектур на основе служб, в которых большая часть семантики хранится в базе данных. Это означает уменьшение количества обращений и улучшение поддержки целостности. Кроме того, наличие единой системы ведет к более естественной и однородной модели данных, т.е. логическая модель базы данных приводится в соответствие с проектным решением библиотеки классов, а необходимость в каких-либо специальных методах анализа данных отпадает.

В объектно-ориентированных базах данных нет необходимости в частых объединениях — самой медленной операции, которую реляционной системе когда-либо приходилось выполнять. Поскольку объекты хранятся в виде когерентных, целостных структур, поиск объекта является единой, простой, хотя, возможно, и большой операцией. В реляционной среде приложение формирует структуру базы данных из отдельных таблиц во время выполнения, в то время как в объектно-ориентированной базе данных хранится непосредственно вся структура. Так, вместо выполнения объединения приложение следует по указателям от объекта к объекту. Это напоминает структуры сетевых баз данных, но объектно-ориентированные базы данных более гибки в смысле эволюции схемы. Это так, поскольку объектное модельное представление означает, что данные хранятся когерентными порциями. Указатели зачастую ограничиваются пределами одного файла. Это также отражает преимущества инкапсуляции, потому что в сетевой базе данных программист должен знать, каким образом реализованы связи между типами записи: в виде связного списка или массива указателей. В зависимости от этого используются различные команды.

Чисто реляционные системы управления базами данных (РСУБД) могут хранить объекты в форме атрибутов или таблиц, но трафик между приложением и такой базой данных увеличивается из-за необходимости повторно "собирать" объекты после поиска и, наоборот, "разбирать" для хранения. Как правило, для представления реального объекта требуется

---

[3] Названо так, поскольку несоответствие импеданса в электрических системах препятствует оптимальной передаче энергии.

много нормализованных таблиц, что ведет к необходимости операций объединения для восстановления когерентных объектов. В особенности это касается сложных объектов, таких как графические изображения, документы, балансовые отчеты и т.д. Эта проблема производительности связана с прежними аргументами в пользу сетевых, а не реляционных баз данных, но теперь эти аргументы могут быть приведены скорее на уровне логических концепций, чем на физическом уровне, где единственными доказательствами являются эмпирические результаты. Типичные РСУБД оптимизированы для нескольких больших объединений, которые характерны для их обычных коммерческих приложений. Кроме того, эти программные продукты настраивались и совершенствовались на протяжении 20 лет. Это делает их более эффективными, по сравнению с современными объектно-ориентированными базами данных (ООБД), при построении систем бухгалтерского учета, управления состоянием товарных запасов и т.п. Однако существуют приложения, которые должны выполнять множество небольших объединений, например, для получения комплексных спецификаций материалов, а также для обработки структурированных документов и данных в географических информационных системах, в приложениях электронной коммерции и в системах автоматизированного проектирования (CAD). Все эти приложения должны хранить сложные объекты, а для этого вида работ обычно наиболее эффективны ООБД.

В ООБД ограничения целостности могут касаться экземпляров, в отличие от реляционных систем, где ограничения могут быть выражены только для сущности (или на уровне класса). В ООБД семантика приложения, например функциональные зависимости, не скрыта в нормализации таблиц. Проблему избыточности решают указатели на объекты.

Реляционные системы обычно поддерживают ограниченный набор примитивных типов данных: целое число, денежная единица, дата и т.д. Некоторые системы поддерживают расширенные типы, например текстовые или большие двоичные объекты BLOB (binary large object), но они не обладают доступной внутренней структурой или семантикой. Объектно-ориентированные и объектно-реляционные базы данных предоставляют возможность введения типов, определяемых пользователем, с семантикой, которая определяется посредством системы классов. Это делает возможным использование объектно-ориентированного языка программирования как в качестве средства доступа, так и в качестве языка манипулирования данными DML.

В объектно-ориентированном программировании имеются два способа определения подклассов: путем объединения типов или явного определения подтипов. Второй способ для систем баз данных более предпочтителен, поскольку в этом случае не разрешается создавать специальные типы из семантически несвязанных типов, и это закономерно распространяется на типы со скрытыми методами.

Приведем критерии качества объектно-ориентированных баз данных.

- Доступность платформы
- Работоспособность и поддержка
- Цена и стратегия лицензирования
- Производительность
- Множественное наследование
- Композиционные структуры
- Динамическая эволюция схемы

- Управление хранением
- Управление блокированием
- Уведомление о событии и его передача
- Управление версиями
- Гибкое управление параллелизмом
- Возможности групповой работы и контроль версий
- Управление запросами
- Безопасность и авторизация
- Интерфейс пользователя и программирование на языках четвертого поколения — визуальное программирование
- Интерфейсы внешнего языка и программные API-интерфейсы баз данных
- Поддержка мультимедиа
- Поддержка Web

Объектно-ориентированные базы данных важны, поскольку они направлены на решение некоторых ключевых проблем применения информационных технологий. К тому же, кроме преимуществ систем баз данных, они обеспечивают преимущества, связанные с объектно-ориентированным программированием, возможностью повторного использования и расширяемостью. При этом можно предложить более широкий диапазон приложений со значительно улучшенной производительностью и экономным использованием памяти (по сравнению с реляционными и гибридными системами) при сохранении их гибкости, подтвержденной опытом. Возможно, важнее всего то, что они обещают предоставить преимущества, связанные с семантически богатыми языками. Это приблизит ООБД к семантическим базам данных и экспертным системам. По мнению автора, объектно-ориентированные базы данных и связанные с ними языки 4GL найдут своих преданных поклонников тогда, когда они будут способны соединить объектно-ориентированное программирование, семантические модели данных и экспертные системы в одной унифицированной среде.

5.7. Преимущества объектно-ориентированных баз данных

Преимущества, ожидаемые пользователями от объектно-ориентированных баз данных, связаны с применением объектно-ориентированного программирования, а именно возможностью многократного использования кода, улучшенной модульностью и расширяемостью. Хочется верить, что при этом сохранится гибкость реляционных систем, которая объединится с эффективностью и семантической выразительностью систем, базирующихся на указателях.

Для наглядности удобно сгруппировать преимущества использования объектно-ориентированных баз данных в три категории: преимущества, связанные с объектно-ориентированным языком программирования или объектно-ориентированной средой; преимущества, определяемые семантически богатыми возможностями объектно-ориентированной базы данных; и преимущества, возникающие непосредственно из возможностей самого объектно-ориентированного проектного решения базы данных.

Все замечания в этом разделе одинаково применимы и к объектно-реляционным, и к объектно-ориентированным базам данных, за исключением тех, которые относятся к производительности.

5.7.1. Преимущества, связанные с объектно-ориентированным языком программирования

Если по каким-то причинам, которые здесь не будут обсуждаться, организация уже приняла решение о необходимости применения объектно-ориентированного программирования и управления перманентными данными, то в результате связывания языка программирования приложения и среды с объектно-ориентированной системой баз данных появляется несколько преимуществ.

- При использовании объектно-ориентированного программирования в объектно-ориентированной базе данных минимизируется необходимость загрузки объектов в память и их удаления из памяти. Остальные преимущества связаны с системами баз данных в целом, но ООБД особенно подходят для совместного использования, обеспечения распределенности, безопасности и управления версиями.
- Объектно-ориентированные базы данных помогают преодолеть семантический разрыв между реальными объектами и понятиями и их представлением в базе данных. Они также уменьшают несоответствие импеданса между языком программирования приложения и языком DML, что облегчает процесс исследования и проектирования за счет унификации процесса проектирования.

5.7.2. Преимущества, связанные с семантически богатыми возможностями

Эти преимущества, в основном, состоят в следующем.

- Возможность хранить как связи между объектами, так и ограничения для объектов преимущественно на сервере, а не в клиентском приложении. Это может иметь огромные последствия как для целостности сопровождения, так и для целостности приложения. Изменение выполняется только один раз, и возможность многократных, несогласованных изменений исключается или, по крайней мере, минимизируется.
- Объектно-ориентированная база данных или модель могут отобразить данные модели "сущность-связь" (иерархию модели, сетевую структуру) и при этом обогатить их семантически. По эффективности они приближаются к иерархическим системам, но динамическое связывание методов и "сборка мусора" могут несколько снизить эффективность доступа.

- В объектно-ориентированной базе данных имеется, по крайней мере, потенциальная поддержка для "правил" (активных объектов). Все современные предложения являются "доморощенными", но можно предвидеть появление стандартного протокола для инкапсулированных правил.
- В объектно-ориентированных базах данных существует возможность предварительного контроля соответствия типов специальных запросов, в отличие от реляционных систем. Это выполняется посредством системы типов, определяемых пользователем, которая может использоваться для отклонения неправильных типов перед выполнением фактического доступа к базе данных. Компромиссом для реляционных систем является предварительная компиляция часто используемых запросов, как это сделано в системе Ingres.

В некотором смысле семантические возможности программирования и системы баз данных — это вещи в себе. Давней целью инженерии программного обеспечения является обеспечение возможности "обратного проектирования" (reverse engineering), т.е. получения из фрагмента исполняемого кода достаточной информации относительно его назначения и функций, чтобы затем переписать его на другом языке или на другой платформе и, если возможно, сделать это автоматически. При современном положении дел возможность такого обратного проектирования почти невозможна. Самое большее, что можно сделать, — это перекодирование отдельных маленьких модулей, причем с большой долей участия разработчика. Хотя и имеются некоторые хорошие автоматизированные инструментальные средства, поддерживающие этот процесс, но даже для работы с такими средствами все еще требуется высокая квалификация. Причиной такого состояния дел является то, что программы не включают в явной форме содержащуюся в них семантику, как, например, технические задания. Поэтому одним из преимуществ объектно-ориентированного подхода автор считает получение возможности обратного проектирования. Если имеется возможность фиксации семантики объектов и кодирования их в явном, читаемом виде (например, в виде правил или структур классификации), то их можно восстанавливать и использовать для генерирования нового кода, который не должен воспроизводить физическую структуру или структуру модуля исходной системы. Это крайне необходимо для создания прототипа на основе спецификации (или одноразового прототипа).

5.7.3. Преимущества самой объектно-ориентированной базы данных

Объектно-ориентированные базы данных объединяют быстродействие сетевой базы данных с гибкостью реляционной. Эталонный тест показал [149] характерное 100-кратное повышение производительности для определенных видов поиска и корректировки базы данных. Большинство объектно-ориентированных баз данных поддерживает динамическую эволюцию схемы, как и реляционные системы. Это означает, что теперь ООБД будут пригодны для реализации ряда приложений, которым прежние СУБД не подходили. Поддержка длинных транзакций с использованием постоянных блокировок и автоматического контроля версий делает эти продукты основой для нового подхода к распределенным вычислениям. Тот факт, что большинство продуктов с самого начала поддерживало распределенное обновление, является еще одним следствием их появления в культурной среде рабочих станций и сетей. Реляционные системы только недавно с трудом получили такую возможность. Кроме того, для описания распределенных систем естественно применяется концепция передачи сообщений.

Чисто ООБД, как правило, обеспечивают лучшую производительность при работе со сложными объектами и сложными связями. Это происходит, главным образом, в результате того, что для хранения нет необходимости разбивать большие объекты на нормализованные таблицы и повторно "собирать" их во время выполнения посредством медленных операций объединения. Это особенно справедливо для приложений, включающих инженерно-техническую документацию и сложную графику. Кроме того, ООБД позволяют сэкономить дисковое пространство, поскольку в реляционных системах необходимо хранить гораздо большее количество индексных файлов. Это единственное преимущество, которое не относится к объектно-реляционным базам данных, хотя тщательная настройка значительно уменьшает это различие для многих приложений.

Объектный подход естественным образом поддерживает мультимедийные приложения. Объекты со свойствами форм, временным поведением (видеосигналы и звук), текстом и другими особенностями можно моделировать и сохранять в рамках унифицированной концептуальной схемы.

Нет необходимости вновь заявлять о выгодах повторного использования. Основным преимуществом является расширяемость, которая вытекает непосредственно из структур классификации объектно-ориентированной базы данных. По мнению автора, возможность повторного использования нельзя причислять к основным преимуществам, поскольку в сложных структурах может нарушаться принцип инкапсуляции. Однако возможность повторного использования потенциально является главным преимуществом объектно-ориентированных языков программирования, связанным с созданием небольшого, четко специфицированного интерфейса между приложением объектно-ориентированного языка и объектно-ориентированной базой данных. Это означает, что преимущество повторного использования больше касается непосредственно самого приложения. ООБД поддерживают расширяемость до тех пор, пока типы данных могут расширяться при помощи своих возможностей наследования. Новые подтипы могут добавляться без необходимости реструктурирования существующих блоков базы данных. Упрощается моделирование агрегирования, а объекты внутри модели могут рассматриваться как отдельные сущности или совокупности объектов.

Развитые объектно-ориентированные базы данных могут упростить управление параллелизмом, а в результате современных исследований может появиться логически правильный метод оптимизации запроса, но это все еще нерешенная проблема.

Как правило, объектно-ориентированные базы данных обеспечивают более эффективное управление запросами, поскольку в объектах хранятся указатели на связанные с ними объекты.

Объектно-ориентированные базы данных обеспечивают встроенную ссылочную целостность и обычно более богатую семантику, чем другие системы. Эти преимущества свойственны системам на основе семантических моделей, модели "сущность-связь" и дедуктивных баз данных. Объектная идентичность решает проблему висячих кортежей[4] и устраняет необходимость проверки целостности ссылок, поскольку на объекты не ссылаются, а обращаются к ним непосредственно. Более богатая семантика базируется на способности создавать типы и инкапсулировать ограничения внутри объектов на любом уровне иерархии типов.

Физическая кластеризация объектов на диске является проблемой для проектировщиков реляционных систем. Для минимизации операций доступа к диску они должны гарантировать, что таблицы, которые часто объединяются, постоянно хранятся на одном и том же запоминающем устройстве или близко друг от друга в памяти. В этом случае единственной

---

[4] Висячие кортежи — это кортежи, которые не представлены во всех (нормализованных) отношениях, связанных с представлением объекта, и которые при выполнении операции объединения могут быть потеряны.

основой для кластеризации является эмпирический анализ общих запросов. В объектно-ориентированных системах имеется более логичная основа для кластеризации, базирующаяся на классах и иерархиях агрегации. Объекты, которые образуют группы, хранятся вместе как составные объекты и в хорошо разработанной структуре классификации используются в качестве естественной основы кластеризации. Это не только обеспечивает удобство реализации; кластеризация отражает семантику приложения и, следовательно, делает вероятным (хотя и не гарантирует) обращение общих запросов к близлежащим физическим участкам на диске.

Отсутствие отдельного языка DML означает, что разработчики должны изучить только один язык. Это возможно благодаря соответствию системы типов в приложении и базе данных. Один из способов осуществления этого состоит в использовании интерфейсов языка Java или классов языка C++ для расширения системы типов с целью включения типов, характерных для языка DML, например типа "отношение".

Общие библиотеки классов приложения и базы данных помогают ускорить разработку. Однако могут возникнуть трудности в обеспечении наличия библиотек и их перемещении. (Эти трудности упоминались в предыдущих главах при анализе объектно-ориентированного программирования.) Для этого весьма важными инструментальными средствами являются броузеры и библиотекари классов. При этом также ценным свойством является возможность групповой работы.

Хотя блокирование проще описать в объектно-ориентированной модели, поскольку естественным модулем блокирования является объект (проще реализовать однократную блокировку всех соответствующих данных, чем многократное блокирование разрозненных несвязанных таблиц), тем не менее в реальных системах блокирование оказалось весьма серьезной проблемой. Как будет видно далее, легче блокировать большие иерархии, но это существенно снижает производительность системы. Конечно, увеличение масштабов блокировок также возможно в реляционных базах данных. Многопользовательский доступ все еще является проблемой в объектно-ориентированных системах, поэтому необходимы реальные подтверждения преимуществ предлагаемых решений.

Объектно-ориентированные базы данных соответствуют распределенным структурам. С распределенной структурой согласуется и передача сообщений инкапсулированным методам объекта, поскольку нет необходимости повторять одну и ту же функциональность во многих приложениях. Эту идею иллюстрирует приведенный в главе 1 пример, где метод вычисления возраста служащего или любого человека хранится вместе с объектом. Не нужно копировать подпрограммы вычисления возраста в несколько приложений, поскольку при их централизованном хранении устраняется опасность параллельного выполнения одних и тех же вычислений.

Автор привел доводы в пользу способности компьютерных систем фиксировать в явной форме как информационную, так и функциональную семантику приложений. Эта тема более расширенно рассматривается в главе 6.

Объектно-ориентированный подход охватывает концептуальную модель, проектирование, программирование и систему запросов. Преимущества унифицированного модельного представления очевидны. Стоит надеяться, что такой стиль приведет к более высокой производительности и гибкости, но в конечном счете это будет зависеть от того, насколько хорошо программисты понимают эту идеологию.

5.7.4. Проблемы объектно-ориентированных баз данных

Как и в отношении других аспектов этой технологии, наряду с преимуществами имеются и недостатки. В этом разделе будут рассмотрены некоторые из них.

На чисто теоретическом уровне возникает проблема отсутствия универсальной общепринятой объектно-ориентированной модели данных. Тем не менее в этом направлении было много предложений. В [765] описана несложная и вполне общая модель. Этому вопросу посвящены также работы [66, 476, 517]. Но эта проблема не должна очень беспокоить технического специалиста-практика. Как уже упоминалось, широкое распространение получили модели комитета CADF и группы ODMG, которые со временем даже могут быть объединены.

Объектная идентичность очень хороша в теории, но реальные объекты не всегда обладают доступной уникальной идентичностью. Это возможно не только в реляционной базе данных, в которой данные об одном и том же человеке могут быть представлены дважды (как сведения об акционере и о служащем). В этом случае в объектно-ориентированной базе данных может существовать два представления одного объекта, обладающих совсем не очевидной уникальной идентичностью. Единственным способом решения этой проблемы является моделирование идентичности ролей, а не объектов, играющих эти роли.

В объектно-ориентированной базе данных невозможно выполнить полную оптимизацию запроса без дискредитации целей объектной ориентации, особенно принципа сокрытия информации, поскольку оптимизация требует просмотра реализации. С другой стороны, запросы по сложным объектам, так или иначе, выполняются намного быстрее. Оптимизацию запроса могут выполнять объектно-ориентированные базы данных, базирующиеся на функциональной модели, впрочем в [341] показано, как это можно сделать в объектно-ориентированном расширении системы Daplex. В [447] автор указывает, что, вероятнее всего, потребуются только небольшие изменения существующих методов оптимизации, но признает, что, по сути, эта проблема все еще находится на стадии исследований.

Еще одна проблема объектно-ориентированных баз данных заключается в трудности моделирования типов связей. Трудно также компоновать операции, поскольку не все они возвращают табличные значения, как в реляционной системе; они могут возвращать значения любого типа. Результат операции может даже относиться к новому абстрактному типу данных.

Потенциальная проблема возникает также из-за множества различных версий множественного наследования, поддерживаемых различными продуктами баз данных и языков. Любому желающему использовать этот аспект объектно-ориентированной технологии необходимо гарантировать, что язык программирования и база данных в этом отношении совместимы.

Как и в случае распределенных реляционных баз данных, распределенные объектно-ориентированные базы данных вызывают проблемы, которые все еще требуют решения. Возникают не только традиционные проблемы выполнения традиционного блокирования и реализации стратегии фиксации (транзакции), но и дополнительные проблемы. Например, никто пока не знает, как гарантировать сетевую объектную идентичность, хотя определенный прогресс достигнут в области разработок на основе компонентов (component-based) (см. главу 9). Любая система, которая использует двухфазные фиксации для выполнения обновления, будет неизбежно медленнее централизованной базы данных. Только параллельные реализации могут решить эту проблему. Кроме того, потенциально непреодолимой проблемой является размещение данных. Практический опыт показывает, что очень редко имеется совершенная стратегия распределения; потребности одного пользователя в локальной памяти для выполнения комплексного анализа могут противоречить требованиям другого, которому необходимо объединять те же данные с другими для регулярных отчетов.

Имеется также несколько других нерешенных проблем, касающихся управления параллелизмом, эволюции схемы, блокирования и оптимизации запросов. Для получения более подробных сведений об этих проблемах и исследованиях, направленных на их преодоление, читатель может обратиться к техническим руководствам, таким как [341, 441, 447, 829], и другим работам, упомянутым в конце этой главы.

Хотя логика протокола блокирования в объектно-ориентированной системе легче для понимания, в многопользовательской среде блокирование не обязательно будет выполняться эффективно. Имеются очень серьезные проблемы блокирования, связанные с иерархиями наследования.

Вопреки рассмотренным преимуществам объектно-ориентированной технологии, остаются проблемы проектирования и реализации. Например, проектировщик все еще должен обдумывать построение индексов, кэширование и т.д. Для этого необходимы квалифицированные специалисты. Кроме того, требуется и хорошее управление. Некоторые из проблем управления будут рассмотрены в главе 9.

Объектно-ориентированные базы данных на сегодняшний день представляют зрелую, но все еще развивающуюся технологию. Все еще пополняется список необходимых требований к этим продуктам. Однако процесс созревания происходит очень быстро. Среди нескольких объектно-ориентированных продуктов баз данных, которые будут рассматриваться ниже, какие-то хороши для одних типов приложения, а какие-то — для других. Если используется Smalltalk и требуется мультимедийная база данных, то можно воспользоваться системой GemStone. Однако для расширения базы данных, пользовательский интерфейс которой написан на C++, более естественным выбором является ObjectStore. Для коммерческого приложения, включающего мультимедиа или сложные объекты, предпочтительнее Jasmine или Versant.

5.8. Обзор программных продуктов ООБД

В данной книге невозможно сделать полный обзор всех существующих объектно-ориентированных продуктов баз данных. Основной целью этого раздела является исследование применения этой технологии, а также перспектив на будущее. Но, как и в случае объектно-ориентированных языков программирования, невозможно избежать краткого обзора. Объектно-реляционные базы данных уже рассматривались выше. Теперь необходимо перейти к современным чисто объектно-ориентированным продуктам и к некоторым из их достойных предшественников.

На рынке имеется три типа продуктов объектно-ориентированных баз данных. Выше уже рассматривались объектно-реляционные гибриды. Чисто ООБД делятся на два вида: те, которые отличаются, главным образом, своей высокой производительностью, и те, которые обладают более низким импедансом несоответствия. Постоянные расширения объектно-ориентированных языков, такие как ObjectStore, обеспечивают способ оперирования перманентностью без импеданса несоответствия, но они обычно довольно слабы относительно традиционной системы **управления** базой данных. Истинно объектно-ориентированные системы управления базами данных, такие как O_2 и Versant, лучше оснащены возможностями систем управления базами данных. Например, O_2 поддерживает язык 4GL, а Versant обладает возможностью уведомления о событиях. В настоящее время оба типа продуктов стремятся к тесной связи с одним из объектно-ориентированных языков (обычно C++ или Java) и к дополнительным связям с другими языками.

5.8.1. Коммерческие объектно-ориентированные базы данных

Gemstone

Одним из самых ранних и, вероятно, самым "чистым" из существующих промышленных продуктов объектно-ориентированных баз данных является GemStone [114, 513]. Продукт GemStone первоначально строился на расширении Smalltalk, известном как OPAL, и предназначался для оперирования перманентными объектами. В отличие от ObjectStore и Versant (см. ниже), которые тесно связаны с языком C++, GemStone не привязан непосредственно к какому-либо конкретному языку, но в настоящее время подчеркивается его связь с языком Java. Таким образом, хотя языком манипулирования данными является OPAL, к GemStone можно обращаться из других языков, включая C++. Однако при этом не поддерживается множественное наследование. Продукт GemStone предназначался, главным образом, для разработчиков, создающих серверы приложений на базе трехуровневой архитектуры клиент/сервер. Он представлен на рынке дольше, чем любой из его конкурентов, и является одним из наиболее законченных программных продуктов.

OPAL имеет библиотеку классов, охватывающую несколько основных, но сложных структур данных, и поддерживает все ключевые возможности объектно-ориентированных систем: абстракцию, наследование и идентичность. Он, как и Smalltalk, по умолчанию является языком, в котором не предусмотрено определение типов данных. В отличие от наиболее общих языков баз данных, даже примитивные действия, такие как вставка, удаление и обновление, должны быть объявлены явно для каждого определенного класса, что неудобно. Это отражает строгую философию инкапсуляции, положенную в основу Smalltalk. К любой системе, базирующейся на языке Smalltalk (с его сборкой "мусора" и принудительным динамическим связыванием), нужно подходить с некоторым скептицизмом относительно производительности. Еще одна небольшая потенциальная проблема состоит в том, что язык доступа GemStone формирует запросы и индексы по атрибутам объектов. Это может послужить причиной нарушения инкапсуляции. Атрибутам и классам можно выборочно задавать типы или **виды**, но вследствие полиморфизма это не защищает от сбоев во время выполнения программы.

Архитектура базируется на модели клиент/сервер и поддерживает распределенные вычисления. Stone — это сервер баз данных или диспетчер объектов, доступ к которому происходит посредством множественных версий Gem — виртуальной машины, которая компилирует и выполняет методы OPAL. Stone отвечает за сборку "мусора". GemStone — это один из немногих продуктов, который хранит методы в базе данных. Благодаря этому и архитектуре клиент/сервер, возможно обращение к GemStone из других языков, включая C, Паскаль и C++. Поскольку методы могут выполняться на сервере, доступ к большим наборам объектов намного эффективнее, так как клиенту возвращаются только результаты. В достаточной степени поддерживаются прозрачность местоположения данных и репликация, что незаметно для пользователя.

Объекты удаляются только с помощью сборки "мусора". Расположение каждого объекта хранится в таблице, что упрощает перемещение объектов. Стратегия блокирования GemStone включает дублирование базы данных для восстановления и комбинацию "оптимистического и пессимистического" режимов для управления параллелизмом. Каждый клиент поддерживает копию объектной таблицы. В случае "пессимистического" режима управления параллелизмом приложения должны получать явные блокировки. В [258] доказано, что "оптимистический" подход приводит к снижению производительности, поскольку требуется больше времени для разрешения конфликтов и т.д. Однако этот подход может быть более

подходящим для средств проектирования и разработки, в то время как "пессимистический" подход мог бы быть удобным для MIS-приложений (Management Information System). "Оптимистическое" блокирование, несмотря на более интенсивное использование центрального процессора (CPU), чем при "пессимистическом" подходе, является все же целесообразным для MIS-приложений вследствие соответствия между производительностью CPU и скоростью доступа к диску на современных машинах. К тому же для запросов только для чтения не требуются какие-либо блокировки, в результате уменьшаются противоречия между обновлениями и транзакциями только для чтения. GemStone поддерживает блокировку для операций чтения, записи и блокировку для обеспечения взаимоисключающего доступа. Блокировки поддерживаются в рамках транзакции и должны быть реализованы явно. Блокировка возможна для индивидуальных объектов или совокупностей неоднородных объектов. Однако GemStone не поддерживает блокирование и копирование совокупностей *взаимосвязанных* (составных) объектов или манипулирование ими.

Объектно-ориентированной базе данных GemStone, как и большинству других ООБД, недостает средств для семантического моделирования данных, таких как структуры агрегирования, исключения, правила, ограничения, триггеры и т.п. Кроме того, ей недостает поддержки многозначных атрибутов отношений, значениями которых являются множества, отношений "один к одному", инверсных отношений и ключей. В [341] подробно рассмотрены эти слабые места объектно-ориентированных баз данных и представлена система ADAM, в которой некоторые из этих недостатков устранены.

Продукт GemStone прост в использовании для программистов, работающих на Smalltalk. В последнюю реализацию GemstoneJ 3.0 интегрированы язык Java 2, серверные страницы Java (Java Server Pages — JSP[5]), процессор Java Servlet, EJB-контейнеры, брокер объектных запросов ORB (Object Request Broker), согласованный с технологией CORBA, и монитор транзакций (Transaction Monitor), разработанный на базе языка Java. Этот программный продукт также соответствует самым современным стандартам и практическим требованиям безопасности.

Versant

Одним из наиболее удачных продуктов объектно-ориентированных баз данных, основанных на объектной технологии Versant Object Technology, является Versant. В качестве первичных языков доступа используются C++ и Java, хотя также пригодны языки C и Smalltalk. Поддерживается также язык SQL, согласованный с языком OQL для поддержки обращения к новой базе данных из существующих приложений. Одной из немаловажных возможностей является уведомление о событиях, базирующееся на обновлении экземпляра, что способствует связи между процессами и групповой работе, а также выполнению приложений, ориентированных на транзакции. Система Versant базируется на многоклиентской/многосерверной архитектуре, подобно системе Ingres. Она получила очень высокую оценку при эталонном тестировании Кателя (Cattell benchmark). Многопоточные, многосеансовые клиенты и многопоточные серверы способствуют более высокой производительности и масштабируемости. Система Versant обеспечивает шлюзы к реляционным системам, таким как Sybase и Oracle, и к другим системам через ODBC (Open DataBase Connectivity — открытый интерфейс доступа к базам данных) и JDBC (Java DataBase Connectivity — интерфейс доступа Java-приложений к базам данных). Существует несколько средств для разработки графического

[5] JSP обеспечивают генерацию динамических Web-страниц, которые отделяют динамическое содержимое от представления. Они компилируются в повторно используемые компоненты — сервлеты (servlets).

интерфейса пользователя и генерации отчетов, а также поддерживается связь с CASE-средствами (Computer-Aided Software Engineering) автоматизированного проектирования и создания программ.

Отказоустойчивый сервер Versant предоставляет возможность круглосуточного доступа через асинхронную репликацию. Динамическая эволюция схемы означает, что новые типы могут добавляться даже при активной базе данных. Эти возможности, наряду с производительностью объектно-ориентированной базы данных, делают систему Versant идеальной для поддержки загруженных Web-узлов.

Другие возможности системы Versant включают хорошую поддержку для композиционных структур, множественного наследования, контроля версий и расширенной библиотеки классов. Чрезвычайно сильные возможности управления транзакциями системы Versant в значительной степени поддерживают совместную работу. Обеспечиваются как оптимистическая, так и пессимистическая стратегии блокирования, и экземпляры могут блокироваться по одному или группами. Двойная стратегия кэширования также увеличивает производительность и масштабируемость. Запросы выполняются на сервере и затем помещаются в буфер клиента. Программисты, использующие технологию CORBA, могут прямо использовать в качестве ссылок на объекты их идентификаторы.

Система Versant использовалась в качестве основы при разработке нескольких активных Web-узлов и приложений в области здравоохранения, передачи данных и управления в условиях финансового риска.

ObjectStore

Система ObjectStore [45] — это еще одна из успешных объектно-ориентированных систем управления базами данных, тесно связанная с языком C++. Большинство рассматривает ее скорее как перманентное расширение языка C++, чем систему управления базой данных. В 1997 году к ней был добавлен свой Java-интерфейс. Как и система Versant, ObjectStore, согласно эталонному тесту Кателя, обладает высокой производительностью и имеет многоклиентскую/многосерверную архитектуру для поддержки распределенных вычислений и масштабируемости. Имеются библиотеки классов для поддержки контроля версий, управления конфигурацией и композитными структурами. Библиотека классов поддерживает также индексирование объектов и кластеризацию, ассоциативные запросы, управление связями и операции над множествами. Существует опасность нарушения инкапсуляции ассоциативными запросами. Отсутствует язык Object SQL. Считается, что устранение импеданса несоответствия и предоставление классов настолько облегчает программирование, что нет никакой необходимости в сложностях языка Object SQL. Система ObjectStore на период написания книги обладала меньшей продуктивностью, чем Jasmine, O_2 или Versant.

Эффективность обеспечивается в результате использования методов переадресации указателей (pointer swizzling technique). Переадресация указателей означает, что глобальные объектные ссылки преобразуются в локальные адреса при загрузке объекта в основную память. Методы переадресации используются для сокращения времени доступа к атрибуту до уровня, свойственного структурам памяти языка программирования. Альтернативный подход заключается в разделении объектного пространства. В этом случае идентификаторы объекта внутри одного сегмента не нуждаются в трансляции. Статическая или динамическая кластеризация связанных объектов на диске также способствует повышению эффективности доступа. Некоторые действия, которые можно выполнять в системе ObjectStore с помощью указателей, например указание на постоянные объекты непосредственно в виртуальной

памяти, кажутся, по мнению автора, немного опасными, но мощность редко достигается без риска. Этот продукт обладает очень хорошими возможностями контроля версий и управления транзакциями, способностью множественного наследования, а также может поддерживать групповую работу. Его уникальный подход к перманентности позволяет напрямую использовать инструментальные средства и библиотеки сторонних производителей. Все методы обрабатываются в клиентской части. Как и система Versant, этот продукт обеспечивает хорошую поддержку целостности ссылок (раздел 5.9).

Система ObjectStore успешно использовалась в качестве объектного кэша для реляционных баз данных. Результатом явилось значительное увеличение производительности и устранение несоответствия импеданса [534]. Это определяет использование продукта в качестве высокопроизводительного, масштабируемого механизма для Web-серверов. Компания Object Design предлагает несколько инструментальных средств повышения производительности, поддерживающих технологии Web, CORBA и язык XML (Extensible Markup Language). Они интегрированы в продукт, включая eXcelon — сервер данных XML, предназначенный для управления существующими данными и их распределения через промежуточный логический уровень между Web-клиентом и базой данных. Приложения ObjectStore включают сетевое управление и серверы приложений, обеспечивающие внешний интерфейс с реляционными программными продуктами.

O_2

Система O_2 корпорации O_2 Technology [51, 222] поддерживает многие объектно-ориентированные языки программирования и свой собственный язык 4GL — супермножество языка C, известное как O_2C. Разработчики O_2 инициировали многие из фундаментальных идей, касающихся объектно-ориентированных баз данных. Система поставляется с графическим броузером, средой программирования, отладчиком и поддерживает язык OQL. Разработка выполняется в режиме интерпретации, в котором возможна эволюция схемы. Конечная система компилируется, и, таким образом, эволюция схемы прекращается. Система O_2 поддерживает индексирование иерархии классов и кластеризацию диска, а также различает значения и объекты. Для эффективности все методы объектов обрабатываются на клиентских рабочих станциях с использованием архитектуры страничного сервера, напоминающей аналогичную архитектуру системы ObjectStore. Система может сохранять сотни гигабайт данных и использоваться для многих коммерческих приложений. В 1997 году продукт был приобретен корпорацией Unidata.

Jasmine

Система Jasmine компании Computer Associates поддерживает несколько сред, сложные объекты и язык ODQL (OQL-согласованный язык запросов). Она обладает мощными возможностями поддержки Web и анимации, а также поддерживает множественное наследование, экземпляры и методы классов, атрибуты и серверные методы. Система Jasmine имеет собственную визуальную объектно-ориентированную среду разработки и библиотеку классов — Studio, которая поддерживает разработку систем почти без написания кода. Эта библиотека прежде называлась JADE. Методы также могут быть написаны на языках C или C++, имеются API-интерфейсы для языков HTML, Active X, Java и Smalltalk. Jasmine также взаимодействует с реляционными базами данных. Система Jasmine интересна, поскольку за ней стоит очень крупный поставщик, обеспечивающий хорошие шансы для успеха в широкой рыночной нише, например в области кэширования сервера приложения.

POET

Система POET от компании Poet Software из Германии является объектно-ориентированной базой данных, которая работает на нескольких платформах. Она тесно связана с языками C++ и Java. Первоначально система выступала в роли промежуточного компилятора C++, добавляющего перманентные классы и другие расширения к языку. Система поставляется с графическим броузером и оптимизатором запросов. Поддерживаются эволюция схемы и управление версиями. Программист работает в большом, разбитом на разделы пространстве виртуальных адресов, доступ к базе данных обрабатывается системой. Система POET развилась в мощную объектно-ориентированную систему управления базой данных, способную конкурировать с системами Versant, O_2 и Jasmine, несмотря на меньшую независимость от языка. Она поддерживает язык OQL и графические средства языка 4GL — Viewsoft. Одной из ее характерных особенностей является акцент на ассоциативное управление и обеспечение возможностей, подобных полнотекстовому поиску. Имеются обычные связи с реляционными продуктами.

Компания Ericsson использовала систему POET для разработки Internet-телефона "Penny-2". Компактная встроенная база данных Java отвечает требованиям группы ODMG для объектов данных Java JDO (Java Data Objects). JDO облегчает доступ к памяти и хранение данных на всех Java-платформах — от серверов до пластиковых карт (smartcard). Это делает ненужным разработку сложного одноразового программного обеспечения. Программист может теперь концентрироваться на написании Java-приложений. Набор программ ассоциативного управления Poet Content Management Suite расширяет основной продукт, обеспечивая возможность администрирования и хранения данных стандартного обобщенного языка разметки SGML (Standard Generalized Markup Language) и XML. С его помощью пользователь может многократно использовать отдельные компоненты, сохранять документы параллельно на различных языках и использовать несколько текущих систем авторизации. Кроме того, имеется возможность редактировать данные через Internet или корпоративную сеть. Данный продукт использовался для обмена данными каталога через Internet. Когда покупатели и дистрибьюторы контактируют в рамках электронной коммерции, они имеют возможность использовать (и не раз) XML-данные, а также обмениваться ими. Дистрибьюторы могут оперативно подавать свои предложения в систему и оставаться доступными, чтобы помочь покупателям.

Objectivity

Среди ведущих производителей аппаратных средств первой фирмой, которая одобрила коммерческую объектно-ориентированную систему управления базами данных, была Digital — главная компания по изготовлению оборудования для вычислительных машин. Система Objectivity/DB [229] также первоначально была связана с языком C++. Эта система поддерживает распределенную обработку, длинные транзакции и контроль версий. Каждый объект имеет структурный идентификатор, поскольку система Objectivity/DB использует иерархическую модель памяти, в которой на верхнем уровне она представляется в виде единой логической базы данных. Эта логическая база данных составлена из множества распределенных баз данных, которые сами состоят из ряда контейнеров. Контейнеры содержат страницы, на которых постоянно хранятся объекты. Модулем блокирования является контейнер, который может состоять из отдельных активных (*hot-spot*) объектов или групп из тысяч объектов. Возможно динамическое изменение этого уровня модульности. Однако перемещение объекта

в памяти требует назначения нового идентификатора объекта OID (Object Identifier) и обновления всех ссылок, хотя это незаметно для пользователя. Это является отступлением от чисто логических OID. Логические OID замедляют работу системы, поскольку во время операции доступа требуется хэширование, и связи становятся, по существу, такими же, как при реляционных индексированных объединениях. Были разработаны специальные возможности для поддержки приложений автоматизированного проектирования (CAD-систем) в области электротехники и механики.

ORION и ITASCA

Статьи о ранней объектно-ориентированной базе данных ORION, в основном, содержат глубокие размышления по общим проблемам. Научно-исследовательский проект ORION, посвященный проблеме человеко-машинного взаимодействия (Man-Computer Communication, MCC), был сфокусирован на эволюции схемы и синтезе описательной семантики и набора правил, которым необходимо следовать для обеспечения непротиворечивости системы типов при добавлении атрибутов и методов к классу или их удалении из класса. Эти правила отличаются от общепринятых в системах баз данных. Схема объектно-ориентированной базы данных является динамической и может эволюционировать следующими способами без необходимости перекомпиляции.

- Изменения в описании объекта: добавление, удаление или корректировка атрибутов или методов.
- Изменения для объектов, представляемых системой: добавление, удаление или изменение идентичности объекта.
- Изменения структур базы данных: добавление, удаление или модификация наследования, композиции, использования или ассоциативной связи.

Эволюция схемы усложняется наличием наследования. Если изменяется суперкласс, то необходимо проверить на наличие зависимостей все его подклассы. Другие структуры (композиция и использование) аналогичным образом усложняют положение дел. Важно обратить внимание на то, что объектная идентичность должна сохраниться при всех изменениях схемы в объектно-ориентированной базе данных. По этой причине предполагается, что изменения схемы будут разумно постепенны. Эта проблема тесно связана с проблемой контроля версий, и большинство объектно-ориентированных баз данных предлагает нечто вроде контроля версий в рамках экземпляра на уровне класса и схемы. Например, в системе ENCORE [256] имеется класс `History-Bearing-Entity`, содержащий такие атрибуты, как `previous-version` (предыдущая версия), `next-version` (следующая версия) и `member-of-version-set` (элемент множества версий), которые могут быть наследованы любым объектом, которому требуется контроль версий. Множество версий — это ассоциация, содержащая все версии объекта.

Хотя система ORION в значительной степени была разработана под влиянием Smalltalk, в модели имеется ряд важных расширений. Поддерживается множественное наследование и имеется три встроенных метода для разрешения конфликтов: правило "разрешено первому", согласно которому для наследования спорного метода или атрибута используется первый суперкласс из списка суперклассов; пользователь может определить выбор; или пользователь может обеспечить наследование обоих свойств с переименованием одного из них. Атрибуты

могут иметь значения по умолчанию на уровне класса. Важность этой возможности будет показана в следующей главе. Хотя множественное наследование поддерживается на уровне класса, экземпляр может принадлежать только одному классу. Еще одним важным аспектом работы ORION является признание того факта, что структуры классификации (AKO) и структуры композиции (APO) часто тесно связаны, и для этой связи была разработана семантика. Обозначения, используемые в ORION для демонстрации этой связи, показаны на рис. 5.19, где корпус транспортного средства принадлежит одному конкретному транспортному средству, которое не может существовать без него. Связи AKO — это линии, исходящие от атрибутов класса. Кроме того, система ORION поддерживает семантику и протокол контроля версий, а также важные идеи относительно блокирования. Поскольку ORION была исследовательской системой, в ней все еще имеются проблемы в таких областях, как безопасность типов и сборка "мусора", но упомянутые выше идеи, по мнению автора, являются основными; многие из них были подхвачены коммерческими преемниками прототипов ORION.

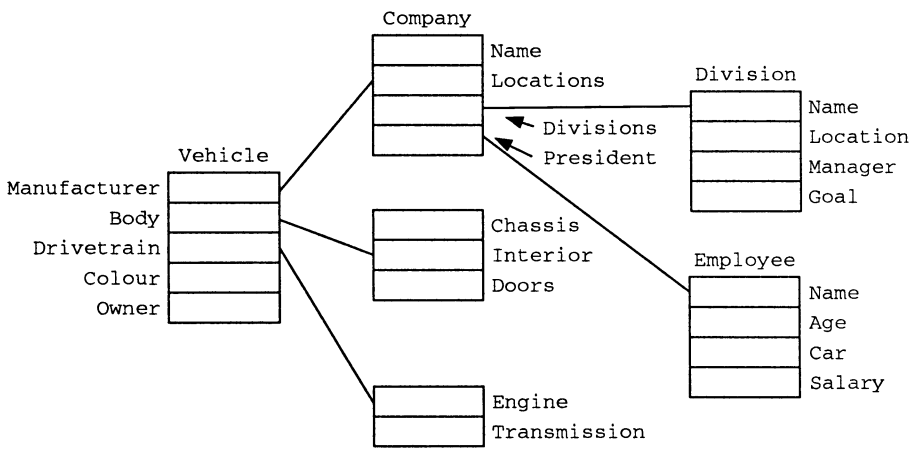


Рис. 5.19. Структура классов системы ORION (публикуется с любезного разрешения ACM Press)

В книге, посвященной ORION [447], приводятся строгие доводы в пользу непроцедурных языков запросов для объектно-ориентированных баз данных, но не исследуется позиция, которой придерживаются многие другие (например, Ульман). Эта позиция сводится к тому, что имеется фундаментальный компромисс между декларативностью и объектной идентичностью. Аргументация [447] относится к базам данных, не приведенным к первой нормальной форме. В определении модели запроса Ким утверждает, что структура объектно-ориентированного запроса "является, в основном, такой, как и структура реляционного запроса". По мнению автора, справедливость этого утверждения зависит от компромиссов, которые были приняты в ORION с целью поддержки непроцедурного подхода к запросам на объединение. С другой стороны, мощность модели запроса не должна игнорироваться. Это целесообразно и даже необходимо для того, чтобы объектно-ориентированные базы данных были конкурентоспособны. В своих исследованиях Ким предлагает содержательную трактовку влияния иерархии (или структуры) композиции и классификации (наследования) на обработку запроса и оптимизацию.

Система Itasca является коммерческим продуктом, базирующимся на исследовательском прототипе ORION. Как и ORION, она была написана на языке Common Lisp. Системе Itasca свойственны все возможности ORION, но она отличается хранением и активизацией методов непосредственно в базе данных, что позволяет встраивать методы, написанные на любом языке. В настоящее время поддерживаются только языки Lisp, CLOS, C, C++, Java и FORTRAN. Itasca базируется на распределенной многоклиентской/многосерверной архитектуре и не имеет централизованного сервера имен или данных, вследствие чего отсутствует единая точка отказа. Поддерживается динамическая эволюция схемы и имеются хорошие службы безопасности, восстановления и управления параллелизмом. Может наследоваться авторизация, что значительно уменьшает трудозатраты на администрирование базы данных. Хорошо поддерживаются структуры композиции, а уведомление о событии может быть реализовано, например, в виде сообщения. Как и в случае Versant, база данных может быть разбита на совместно используемые части и различные персональные базы данных, при этом пользователю нет необходимости знать, где физически хранится конкретный элемент данных. Таким образом, Itasca обладает прозрачностью в отношении размещения данных. Для повышения производительности применяется индексирование классов, а запросы по возможности выполняются параллельно на различных машинах сети. Для оптимизации запроса используется статистика базы данных, подобно оптимизатору системы Ingres. Также возможен ассоциативный полнотекстовый поиск. Динамические возможности продукта делают его пригодным для таких приложений, как разработка данных и их хранилищ, где он занимает свою рыночную нишу среди других ООСУБД.

Очевидными значительными недостатками системы Itasca являются отсутствие возможностей реализации бизнес-правил и ограничений, за исключением методов или триггеров, отсутствие инструментария для языка 4GL и некоторые проблемы с производительностью. Автор считает, что реализация на языке Lisp ограничивает производительность, хотя поставщики однажды уверяли его, что "какое-либо замедление является результатом богатых возможностей продукта, а не языка Lisp".

В результате сравнительного анализа объектно-ориентированных баз данных в контексте инженерных приложений в [12] сделан вывод, что Itasca обладает лучшими возможностями, чем любой другой из исследованных продуктов, и предоставляет самое оптимальное, что можно получить за деньги. Авторы [12] пришли к заключению, что система Itasca обладает "очень широкими возможностями и относительно проста в использовании для программистов на языке Lisp, хотя среда Lisp несколько снижает эффективность работы системы". Однако, несмотря на хорошие отзывы, Itasca не достигла такого коммерческого успеха, как другие упомянутые выше объектно-ориентированные базы данных.

5.8.2. Другие важные продукты и проекты

Vbase и Ontos

Система Vbase фирмы Ontologic была одной из первых настоящих объектно-ориентированных баз данных. Она работает под управлением центрального устройства управления CLU (Central Logic Unit) и ориентирована на строгий контроль типов данных. В ней также была предпринята попытка максимизации связывания во время компиляции. В системе Vbase имеется собственный частный язык определений типов TDL (Type Definition Language) и применяется объектный процессор языка C (COP — C Object Processor) — объектно-ориентированное расширение C, используемое для доступа к данным. Ее уход с

рынка и последующая замена новым продуктом Ontos указали на ряд характерных проблем ООБД первого поколения. Во-первых, в мире коммерческих баз данных был так хорошо принят и понят язык SQL, что введение нового языка, такого как TDL, хотя это была и неплохая идея, было обречено на провал. Более того, на рынке не было места для еще одного объектно-ориентированного расширения C, не говоря уже о столь частном расширении, как COP. Во-вторых, что является более значительным, ранние пользователи Vbase считали, что ее механизм блокирования вызывает серьезные проблемы, касающиеся производительности. В любой системе, которая поддерживает наследование, это потенциально слабое место. Например, в SACIS конкретный элемент принадлежит классу `ModelCar`, который, в свою очередь, является подклассом `Toy`, `Car`, `Stockltem` и т.д. Этот элемент наследует поведение и данные из вышестоящих классов. Допустим, один из этих классов подвергается модификации. Несомненно, что все классы, находящиеся в иерархии ниже него, должны быть блокированы. Если модифицируемый класс является в достаточной мере общим, то можно, в конце концов, блокировать всю базу данных. В многопользовательской базе данных это совершенно неприемлемо. Эта трудность не является исключительно проблемой Vbase (это можно отнести и к несколько ранним объектно-ориентированным базам данных). Фактически аналогичная проблема имеется и в реляционных системах, поддерживающих триггеры для контроля целостности, в которых неконтролируемые модификации могут распространяться через базу данных, и при этом каждая из них будет выполнять блокирование.

Для преодоления или, по крайней мере, ограничения этих трудностей корпорация Ontologic (переименованная Ontos Inc.) выпустила продукт второго поколения Ontos, который также главным образом предназначен для рынка инженерных и CASE-технологий. В Ontos язык TDL был заменен объектно-ориентированным расширением языка SQL, в которое был добавлен объектный синтаксис запроса. Объектный процессор COP был заменен языком C++. Проблема блокирования была решена путем передачи управления стратегией блокирования проектировщику. Это позволяло ему выбрать между пессимистическим и оптимистическим протоколами. Кроме того, производительность была увеличена с использованием кластеризации диска и методов кэширования. Были также добавлены улучшенная возможность уведомления о событиях и обнаружение глобальной тупиковой ситуации, расширенные средства повышения производительности и триггеры. Очень важной возможностью системы Ontos в частности и этого типа продуктов вообще является возможность ведения хронологии для версий объектов и совместного использования транзакций. Ontos был одним из наиболее успешных (в коммерческом смысле) и важных ранних продуктов. В 1993 году компания Ontos Inc. заявила о привлечении более чем 400 активных пользователей. Тем не менее с этого времени компания решила сосредоточиться на поддержке библиотек объектов и приложений в среде Microsoft, и продукт исчез из виду.

IRIS, OpenODB, Odapter и PCLOS

Система IRIS [273] — это еще один продукт, который предложил язык Object SQL (OSQL). Она была разработана как экспериментальная система в компании Hewlett-Packard, и ее модель данных базировалась скорее на функциональном, чем на строго объектно-ориентированном подходе. По стилю она очень близка реляционным базам данных и испытывает сильное влияние со стороны языка Prolog. Лежащий в основе модуль управления памятью сходен с модулем, используемым в системе System R, предшественнице DB2. Созданные в ней объекты могут храниться в реляционных базах данных, включая Oracle, Sybase и HP/ALLBASE. Поддерживается множественное наследование, и тип объекта

может изменяться даже во время выполнения. Это мощная, но опасная возможность. Также поддерживается контроль версий. В дополнение к Object SQL, поддерживается доступ к данным с помощью языков Objective-C и Lisp. Поддержка мультимедийных баз данных в IRIS выполняется через специализированные программы управления данными, а не за счет хранения всех видов объектов в единой системе, как в других ООБД. Поскольку администратор транзакций IRIS в своей основе такой же, как в System R, блокирование иерархии наследования является проблемой. Подход системы Itasca, заключающийся в наличии специальных протоколов блокирования для сетей наследования, гораздо лучше. Коммерческая версия IRIS была выпущена под именем Open ODB [11]. Позже она была улучшена и переименована в Odapter. В этом продукте возможна привязка к нескольким языкам, включая C++, Smalltalk, Java и CORBA IDL. Интерфейсы языков C++, Smalltalk и Java обеспечиваются библиотеками классов, которые включают поддержку различных операций базы данных. Пакет, состоящий из клиента Java, конструктора классов Java и сервера Odapter, продается в виде отдельного продукта, известного как DEPOT/J.

Система PCLOS (Persistent CLOS) была попыткой построения интерфейса между объектно-ориентированной базой данных (IRIS) и объектно-ориентированным языком программирования (CLOS).

Statice

Продукт Statice [743] является расширением среды программирования Genera для машин Symbolics и языка Common Lisp. Этот продукт поддерживает определение объектов с множественным наследованием, объектную идентичность и инкапсуляцию методов. Объекты являются перманентными, и к ним возможен совместный доступ. Замечания относительно производительности систем, базирующихся на языке Lisp, высказанные по отношению к системе Itasca, применимы и к Statice. Однако если необходим язык Lisp, то реализация наиболее предпочтительна на платформе Symbolics, которая оптимизирована для этого языка. Система Statice была также адаптирована к различным рабочим станциям, работающим в операционной среде UNIX.

Другие продукты

В мире объектно-ориентированных баз данных имеется несколько языков, объединяющих принципы объектно-ориентированного программирования и манипулирования данными. К ним относятся Vision и язык компании DEC Trellis/Owl. Язык Vision от компании Innovative Systems Techniques (Insyte) задумывался как достаточно простой в использовании для пользователей в области финансового планирования и поддержки принятия решений. Зависимости данных в приложениях такого вида основаны на времени и временны*х* связях. Для эффективного решения такого вида проблем в Vision выбран функциональный стиль, и, хотя Vision не является настоящей объектно-ориентированной базой данных, это предоставляет возможность использования наследования и временны*х* связей. В Vision применяется понятие объектов-прототипов, которое размывает различие между экземплярами и классами. Vision использовался в финансовых приложениях, в частности в задаче управления пенсионным фондом.

Система G-Base является одной из ранних объектно-ориентированных баз данных. Она была разработана во Франции Графелем (Graphael) и базировалась на языке Lisp. G-Base оснащена Prolog-подобным языком G-Logis, предназначенным для разработки экспертных систем. Она также может быть интегрирована с ART.

Система ODE [10] — это исследовательская система, связанная с языком C++ точно так же, как и ObjectStore. Она поддерживает контроль версий, итерации, обработку запросов, ограничения и триггеры с задержкой. В системе ODE для работы с данными используется процедурное расширение C++, называемое O++ и подобное COP в Vbase.

Система LOOM [431] представляет альтернативное решение проблемы создания единого перманентного пользовательского языка. LOOM эффективно превращает жесткий диск в огромную виртуальную память для Smalltalk. Имеется несколько подобных продуктов, расширяющих Smalltalk.

Кроме упомянутых выше продуктов, был разработан ряд исследовательских систем, повлиявших на более поздние продукты. К ним относятся ObServer/ENCORE [256, 717, 830, 831], OOPS [685] и PDM [517]. SAMOS была одной из действующих объектно-ориентированных баз данных, созданной в исследовательских целях Клаусом Дитрихом (Klaus Dittrich) и его сотрудниками в Университете Цюриха (она и по сей день используется). В Университете Брауна был разработан универсальный объектный сервер Observer. ENCORE (Extensible and Natural Common Object Resource — расширяемый и естественный общий объектный ресурс) является его языком определения данных DDL и языком манипулирования данными DML. Эта система обладала многими из возможностей системы ORION и улучшенным администратором транзакций для поддержки групповой работы. Эти системы являлись испытательными стендами для новых идей и решения проблем.

Наряду с объектно-ориентированными базами данных начали появляться базы данных нового поколения, которые Парсайе назвал "интеллектуальными" [618]. Корнями этих разработок являются различные попытки объединения технологии баз данных с идеями из области искусственного интеллекта и особенно экспертных систем. Часть ранних работ по разработке системы Postgres [730] (некоторые из них были использованы в Illustra) соответствует этой традиции. Было предпринято несколько попыток достижения этой унификации с помощью размещения препроцессоров языка Prolog в базах данных с целью обогащения семантической выразительности языка SQL. Дедуктивные базы данных, которые упоминались в этой главе ранее, также объединили работы по экспертным системам и теории баз данных.

В настоящее время возможными способами достижения возможностей экспертных систем в приложениях, требующих переработки большого количества данных, является использование довольно ограниченных возможностей представления знаний в системах, подобных Illustra, применение дедуктивной базы данных или формирование слабого связывания между базой данных и некоторой оболочкой экспертной системы. Проблемы с таким слабым связыванием возникают при интенсивном обмене информацией между приложением и базой данных. Если, вместо генерирования обращений к базе данных на языке SQL, оболочка загружает для обработки целые сегменты базы данных, то возникают проблемы целостности и управления параллелизмом. Таким образом, приложения, основанные на знаниях или очень частом доступе к базе данных, должны обратить внимание на появление этих "интеллектуальных" баз данных. Формирование такого связывания сопряжено с большими усилиями и даже страданиями.

Многие исследования выполняются в традициях концептуального моделирования и функциональных баз данных. В [341] описаны две системы, P/FDM и ADAM, принадлежащие к данной области. Более современные данные можно получить по запросу из университета Абердина.

5.9. Целостность ссылок в объектных базах данных

В стандарте для объектно-ориентированных баз данных ODMG-93 термин "взаимосвязь" (relationship) используется как синоним для ассоциации (association). В этой книге мы будем придерживаться термина "ассоциация". Стандартная: объектная модель по предложению группы ODMG поддерживает ассоциации "один к одному", "один ко многим" и "многие ко многим". Они определяются как *пути обхода* (traversal path) объектов и объявляются в определениях открытых интерфейсов объектов. Ключевое слово `inverse` в обоих объявлениях пути обхода указывает, что они относятся к одной и той же ассоциации. Данное ключевое слово гарантирует, что ассоциации поддерживают целостность ссылок, которая обеспечивается ОСУБД автоматически. Если ключевое слово `inverse` отсутствует, значит, прохождение в обратном направлении невозможно.

В стандарте ODMG-93 ассоциации считаются типами, которые не обладают объектной идентичностью OID (Object Identity) и уникально идентифицируются с помощью участвующих в них экземпляров. Тип ассоциации "многие ко многим" составляется из двух наборов ассоциаций "один ко многим". Эти ассоциации "один ко многим" в свою очередь определяются наборами ассоциаций (указателями) "один к одному". Для одного направления обхода ассоциации "один ко многим" определяется единственная функция обхода, обеспечивающая переход от одного объекта к набору связанных с ним объектов. Однако функция обхода, определенная для каждого компонента ассоциации "один к одному", позволяет перейти от любого элемента набора назад к одиночному экземпляру.

Для иллюстрации поддержки целостности ссылок по стандарту ODMG-93 автор предлагает рассмотреть в качестве примера систему ObjectStore, поскольку конструкция инверсной ассоциации ObjectStore удовлетворяет стандарту ODMG-93. Автор следует представлению, данному в руководстве по ObjectStore [608].

Система ObjectStore включает и язык программирования объектно-ориентированных баз данных, и систему управления, которая с 1990 года распространяется корпорацией Object Design Inc. Модель данных ObjectStore близка к модели данных C++. ObjectStore может использоваться тремя различными способами.

- С использованием программ на языке C, вызывающих библиотечные функции C.
- С использованием программ на языке C++, вызывающих библиотечные функции C++, с параметризованными классами или без них.
- С использованием ее собственного расширения C++, языка DML, который, в дополнение к конструкциям C++, обеспечивает поддержку параметризованных классов (на основе шаблонов C++), средства организации режима запросов и обработчики исключительных ситуаций.

Последний способ рассматривается здесь в качестве подхода к реализации механизма поддержки целостности ассоциаций, обеспечиваемого в системе ObjectStore, хотя в версии 4 для тех же целей был предусмотрен библиотечный интерфейс.

Необходимо иметь представление о том, каким способом ObjectStore определяет некоторые из своих классов коллекций. Коллекция — это объект, используемый для группирования других объектов в виде упорядоченных или неупорядоченных наборов, в которых допускаются или не допускаются дубликаты.

- `os_Set`: неупорядоченная коллекция без дубликатов
- `os_Bag`: неупорядоченная коллекция с дубликатами
- `os_List`: упорядоченная коллекция с дубликатами

Классы коллекции содержат функции-члены для вставки, удаления и поиска элементов.

В собственном расширении языка C++, реализованном в ObjectStore, ассоциации представляются как элементы данных в общем интерфейсе класса. Эти связи могут представлять ассоциации "один к одному", "один ко многим" и "многие ко многим". Они реализованы в виде бинарных ассоциаций элементов данных, значениями которых являются указатели или коллекции указателей. Использование коллекции в объявлении `inverse_member` определяет кардинальность (количество элементов) ассоциации (cardinality of the association).

ObjectStore предлагает конструкцию и механизм автоматического обеспечения ограничений целостности. Элементы данных могут быть объявлены инверсными друг другу (ключевое слово языка OODM `inverse_member`), чтобы модификация одного элемента данных автоматически вызывала соответствующую модификацию его "инверсии".

Ограничения целостности, лежащие в основе объявления `inverse_member`, интерпретируются здесь в немного измененной версии (для удобочитаемости).

> Если *a* и *b* являются инверсными элементами данных (представляющими ассоциацию), то, для любых двух экземпляров *x* и *y* этих двух классов, *x* представляет собой значение *a* (которое является элементом данных в открытом интерфейсе экземпляра *y*) тогда и только тогда, когда *y* является значением *b* (которое является элементом данных в открытом интерфейсе экземпляра *x*).

В ассоциации "многие ко многим" значением элемента данных есть коллекция, группирующая указатели. Таким образом, строго говоря, каждый элемент данных однозначен. Поскольку ObjectStore поддерживает классы коллекций, в которых допускаются дубликаты (`os_List`, `os_Bag`), необходима более общая версия ограничения целостности. Пример ассоциации "многие ко многим" приведен в табл. 5.1.

> Если *a* и *b* являются инверсными элементами данных (представляющими ассоциацию), то, для любых двух экземпляров *x* и *y* этих классов и для любого целого *n*, *x* встречается *n* раз как значение *a* (которое является элементом данных в общем интерфейсе экземпляра *y*) тогда и только тогда, когда *y* встречается *n* раз как значение *b* (которое является элементом данных в общем интерфейсе экземпляра *x*).

Таблица 5.1. Пример ассоциации "многие ко многим"

Класс S		Класс T	
Экземпляр	**Инверсная ассоциация a**	**Экземпляр**	**Инверсная ассоциация b**
y	a.(x, x, z, r)	x	b.(y, y, s, t)
s	a.(x)	z	b.(y)
t	a.(x)	r	b.(y)

Объявление inverse_member неявно содержит правило о необходимости введения ограничения целостности, как упоминалось выше. Это согласуется с предложением автора [333] по наложению ограничения целостности путем инкапсуляции правил в объявлениях классов, соответствующих ассоциациям, которые представляются как указатели со строгими квазиинверсными правилами.

В листинге 5.1 элементы данных employers (экземпляры класса companies) и employees (экземпляры класса people) сгруппированы в коллекции классов (класс os_Set), ссылающихся друг на друга посредством ключевого слова inverse-member. При выборе другого подхода, отличного от подхода с использованием расширения ObjectStore для C++ (язык DML), приведенный выше пример можно представить также на языке C++. Концепция ассоциации, как таковая, в C++ не поддерживается. Однако она моделируется путем поддержки автоматически генерируемых методов C++ (система предопределенных макросов), которые ее реализуют. Автор не будет давать здесь подробные объяснения.

Ограничения целостности обеспечиваются в следующих случаях.

- Установка значения для ассоциации (если x является значением a для y, y является значением b для x).
- Отмена установки значения для ассоциации (если x отменяет установку значения a для y, y отменяет установку значения b для x).
- Удаление экземпляра (если x удаляется, то отменяется установка значения a для y).

Листинг 5.1. Использование ключевого слова inverse_member системы ObjectStore

```
class people
{public:
 people   *spouse   inverse_member   spouse; //   1-1
 os_Set<companies*>employers  inverse_member  employees; //n-m};

class  companies
{public:
os_Set<people*> employees  inverse_member  employers; //n-m};
```

Обеспечение ограничений целостности внутренне реализуется с помощью класса отношения (relationship class), встроенного в класс, содержащий ассоциацию, инкапсулирующую указатель (коллекцию указателей) на элемент данных. Если не используется элемент языка ObjectStore DML inverse_member (который является макросом), то могут применяться классы связи и классы коллекции из библиотеки классов. Реализация класса отношения полностью скрывает задачи инверсной поддержки от кода, который управляет экземплярами класса, содержащего ассоциацию.

Элемент данных ассоциации в ObjectStore ссылается на свою "инверсию" в открытом интерфейсе другого класса. Таким образом, инкапсуляция не нарушается.

СУБД ObjectStore поддерживает целостность автоматически. Объявление inverse_member гарантирует, что ограничения целостности (которые могут рассматриваться в качестве правил или инвариантов класса, неявно содержащихся в инверсной ассоциации) будут введены в действие. Это похоже на хорошее решение рассмотренной выше проблемы ассоциаций: представление ассоциаций как отображений, которые реализованы в виде указателей

и связаны с правилами для наложения ограничений целостности. То же можно реализовать и в других объектно-ориентированных базах данных, таких как O_2 [222]. Однако для этого потребуется написать некоторый код. В любом случае инкапсуляция не поддерживается реализацией, которая зависит от глобальных свойств языка DML, а не от правил, инкапсулированных непосредственно в классах базы данных.

В других объектно-ориентированных базах данных, таких как O_2, Jasmine и Versant, правила целостности моделируются аналогичным способом. Например, как показано в листинге 5.2, в системе Versant это сделано с использованием ключевого слова `BiLinkVstr` вместо `inverse_member` в ObjectStore.

Возможности повторного использования, которая считается одним из главных преимуществ объектно-ориентированных методов, тоже ничего не угрожает, поскольку если один из классов с объявлением `inverse-member` используется повторно, во время компиляции СУБД обеспечит также импортирование соответствующего класса. Таким образом, информация ассоциации не будет потеряна.

Листинг 5.2. Построение правил целостности в системе Versant

```
BiLink_to_BiLink<x,r>::add() ;          // один к одному
BiLink_to_BiLinkVstr<x,r>::add();       // многие к одному
BiLinkVstr_to_BiLink<x,r>::add();       // один ко многим
BiLinkVstr_to_BiLinkVstr<x,r>::add() // многие ко многим
```

5.10. Приложения объектно-ориентированных баз данных

К приложениям, в которых наиболее предпочтительно применять объектно-ориентированные базы данных, относятся те из них, для которых реляционные системы малопригодны, или те, где удобно применить объектное представление. К таким приложениям относятся распределенные системы и системы на основе архитектуры "клиент/сервер", автоматизированные системы документооборота и моделирования развития предприятий, мультимедийные базы данных, системы голосовой почты, геоинформационные системы GIS (Geographic Information System) и системы управления выпуском продукции. Объектно-ориентированные базы данных могут оказаться полезными для приложений, предназначенных для обработки сложных счетов, связанных с материально-техническим обеспечением.

Ранее были широко распространены классические объектно-ориентированные базы данных, используемые в приложениях по автоматизированному проектированию (CAD — Computer-Aided Design), в том числе по проектированию сверхбольших интегральных микросхем (VLSI — Very Large-Scale Integration). Из-за несоответствия реляционных систем выдвинутым требованиям, в этих приложениях использовались собственные файловые системы, что приводило к удорожанию разработки и невозможности совместного использования данных в нескольких системах.

В типичных CAD-приложениях, где необходимо хранить сложные объекты, представляющие компоненты проекта и логические связи между ними, объектно-ориентированная база данных позволяет повысить производительность, оптимизируя физическое расположение зависимых данных. Если весь проект хранится в виде одиночного объекта, то уменьшается

количество операций доступа, необходимых для просмотра или модификации проектных решений. Разработка программного обеспечения с использованием CASE-средств — это еще одна область, где важна поддержка сложных объектов и связей между ними.

Корпорация IBM использовала систему GemStone для разработки базы данных для сложного производственного процесса. Системы POET, O_2, ObjectStore и Versant использовались при управлении телефонными сетями, а также в системах автоматизированного проектирования, географических информационных системах, системах автоматизации делопроизводства, резервирования авиабилетов, автоматизации документооборота, системах для выявления телефонного мошенничества, управления финансовыми рисками и торговлей, работающих в реальном масштабе времени. Другие приложения, о которых стало известно из различных источников, связаны с планированием требований к материалам и, в меньшей степени, с задачами искусственного интеллекта.

Для этих приложений общим является то, что все они связаны с решением проблем, затрагивающих очень сложные объекты. При этом связи между объектами, как во времени, так и в пространстве, являются столь же важными, как и их отдельная структура. Для этих объектов характерна не только структурная сложность, но и сложность поведения. Например, технический чертеж можно представить как план, вертикальную проекцию, вид сбоку или изометрическую проекцию. Кроме того, эти представления могут поворачиваться и масштабироваться. Возможность сохранения поведенческих абстракций, одно из ключевых свойств объектно-ориентированных баз данных, очень важна для таких приложений. При решении задач, связанных со сложными типами данных, достаточно воспользоваться объектно-реляционными базами данных. Однако если используются сложные сети таких типов, то более пригодными оказываются объектно-ориентированные базы данных.

Наиболее важным приложением объектно-ориентированных баз данных во время написания этой книги являлась поддержка больших и сильно загруженных Web-узлов. В качестве примера можно привести использование системы Versant для поддержки Web-узла корпорации Microsoft, который является одним из наиболее посещаемых. Система ObjectStore также нашла своих потребителей в этой области благодаря серверу данных eXcelon, основанному на использовании языка XML. Средство управления содержимым системы POET также, по существу, является XML-сервером и используется в электронных технических руководствах на основе Web-технологий. Продукт GemStone тоже сфокусирован на обеспечении серверов приложений с высоким уровнем обеспечения целостности.

Система Versant играет ключевую роль в объектно-ориентированной архитектуре системы служб ведущего инвестиционного банка Dresdner Kleinwort Benson. Она также использовалась в банке Chase Manhattan для хранения изменчивой финансовой информации. Эти данные необходимы для комплексной системы ценообразования и финансовых оценок, но их структура достаточно сложна. Изменчивую информацию можно представить в виде плоского или трехмерного графа, а такие структуры данных, конечно, не поддерживаются реляционными системами. Система Versant обеспечила необходимую производительность, и группа разработчиков сообщила, что отсутствие несоответствия импеданса существенно уменьшило ресурсоемкость системы.

Можно возразить, что основной задачей является не обратное проектирование, а возможность корректной, бездефектной разработки, базирующейся на совершенных методах анализа и проектирования. Этот аргумент не лишен смысла, но основан на довольно идеалистическом предположении, что компания никогда не пожелает изменить реализацию системы или отказаться от решений, принятых на стадии анализа.

5.10.1. Распределенные базы данных и поиск информации

Объектно-ориентированные базы данных хороши в качестве распределенных баз данных, но некоторые современные реляционные продукты тоже способны работать в сети. Язык SQL может применяться для соединения клиентов с серверами, но он обычно используется таким способом только для нестандартных запросов. Более сложные приложения предусматривают наличие на сервере связанного с ними кода, обычно в виде хранимых процедур, которые написаны не на стандартном языке SQL, а на его процедурном расширении. Стандарты программных API-интерфейсов, представленные такими продуктами, как Open Server корпорации Sybase, или открытым интерфейсом ODBC фирмы Microsoft, упрощают реализацию систем клиент/сервер с применением реляционных баз данных. Используемая в этих случаях модель — это модель одиночного сервера. Действительно распределенные базы данных поддерживаются все же не очень хорошо. Прозрачность в отношении размещения данных, возможность устранения отказов или неисправностей узла, обработка ошибок и время ожидания — все это серьезные практические проблемы. Данные из иных (не реляционных) баз данных часто могут передаваться в реляционные базы через шлюзы, но интерфейс к этим шлюзам и другим реляционным системам обычно является зависимым от применяемого API-интерфейса. С другой стороны, стандарты, подобные интерфейсам ODBC, JDBC и EJB, обязывают поставщиков баз данных изменять клиентскую часть своих продуктов таким образом, чтобы все клиентские приложения могли общаться с любой базой данных. Однако этот подход можно применять только в случае однопользовательских рабочих станций, а потребность в поддержке многопользовательских клиентов приводит к использованию технологии брокеров объектных запросов.

В [442] утверждается, что решение заключается не столько в использовании брокеров объектных запросов ORB (Object Request Broker), сколько в комбинации объектно-ориентированных, реляционных и так называемых "интеллектуальных" баз данных, основанных на правилах. Авторы этой работы предлагают включать в такие базы данных возможность поиска, обычно связываемого с библиотечными системами извлечения информации IR (Information Retrieval). Наличие полей BLOB или memo-полей, содержащих информацию сложной структуры, подтверждает необходимость в этом расширении. Более того, если в базе данных хранится не только текстовая и числовая информация, но и визуальные и звуковые данные, то проблема индексирования и поиска становится более насущной и сложной. Пользователи должны иметь возможность доступа к содержанию хранимого объекта без его чтения или полного просмотра.

IR-системы, которые оперируют редко изменяемыми документами, могут использовать инвертированную индексацию, смысловую сортировку и средства тезауруса. В более изменчивых приложениях можно сканировать системные документы и сравнивать их содержимое с набором ключевых слов или других лексем. Еще один подход заключается в формировании таблицы сигнатур для каждого документа путем хэширования ключевых слов в маленькую числовую сигнатуру, которая затем может сравниваться с сигнатурой произвольного запроса. Это вероятностный подход, который не гарантирует, что все соответствующие документы будут найдены или что все найденное будет соответствовать запросу. Для реализации идеи системы индексов требуются специалисты с хорошим знанием предметной области, которые смогут подготовить таблицы ключевых слов, списки синонимов, тезаурус и т.д. В работе [442] содержится краткое, но достаточно детальное введение в системы информационного поиска в контексте общих систем автоматизации делопроизводства с учетом поиска не только текстовой, но и звуковой и графической информации. В ней также имеются достаточно подробные

сведения относительно различных технических подходов к детализации данных, сбору данных, структуре синтаксического анализа и понимания, языкам запросов на основе экспертных систем, индексированию и сжатию.

Важной проблемой является смысловой контекст информации. Поэтому разработчики IR-систем сталкиваются с теми же проблемами, что и создатели систем искусственного интеллекта, предназначенных для решения задач понимания естественного языка. Индексирование, сделанное вручную, является обычно слишком дорогим, чтобы быть достойным рассмотрения, поэтому некоторые специалисты исследовали автоматические системы индексирования. В [770] поддерживается использование статистических методов и предлагается рассматривать восстановление информации как процесс доказательства теоремы с небулевой логикой. Автор при этом ссылается на работу [226] по ситуационной теории, в которой определена минимальная единица информации — *infon*.

Особенно важным приложением этой теории в контексте данной книги является ее использование при поиске данных в архиве (repository). Хранение информации — это относительно простая проблема. Главной проблемой IR-систем является поиск необходимой информации. Она имеет особенное значение в объектно-ориентированных системах из-за необходимости поиска информации о библиотеках классов. Имеется два основных предложения по ее решению. Можно организовать ссылки в виде иерархии и для получения доступа использовать структуру классификации или индексировать все спецификации класса по ключевым словам для организации поиска. Ни один из подходов не является идеальным. Успешного повторного использования библиотечных объектов и спецификаций можно достичь только через поиск, основанный на комбинации классификации и ключевых слов. Кроме того, можно расширить оба подхода. Например, при использовании тезауруса IR-системы пользователю при поиске можно предоставить возможность выбора ключевого слова или спецификации класса, а затем предложить возможность повторного поиска с использованием более узкого или расширенного значения термина. Для описания структуры и семантики информации, наряду с семантическими сетями, могут использоваться объектные модели или более мощные расширенные объектные модели, предполагающие включение в объекты наборов нечетких правил (fuzzy ruleset) и использование нечеткого наследования. SOMA-модель расширяет идею иерархического поиска с использованием четырех ортогональных структур и допускает нечеткое соответствие ключевым словам. Эти вопросы рассматриваются ниже в главе 6 и приложении А.

5.11. Соображения стратегии

Итак, учитывая преимущества объектно-ориентированных баз данных, как же поступить — выбрать язык OQL и полностью объектно-ориентированный продукт базы данных, такой как Jasmine, Versant, ObjectStore или POET, или выбрать расширенную реляционную систему, наподобие DB2, Illustra или Oracle?

В действительности предлагаемые варианты не являются единственно возможными. Семантические или дедуктивные базы данных обеспечивают многие из преимуществ объектно-ориентированных систем, и на сегодняшний день существуют и используются реализующие их коммерческие продукты. Компании могут также продолжать использовать свои стандартные реляционные продукты и встраивать в базовый язык объектно-ориентированные конструкции, используя такой интерфейсный программный продукт, как Persistence. Но этот подход

имеет недостатки, часть из которых рассматривалась выше. Можно выбрать специализированную, встроенную в систему базу данных, наподобие используемой в системе CAD/CAM или CASE, и систему управления, поддерживающую групповую работу.

Однако для большинства компаний основным вопросом является выбор между объектно-реляционными и настоящими объектно-ориентированными системами. В последнем случае проводятся различия между расширениями перманентных языков, такими как ObjectStore, и подлинными системами **управления** базами данных, такими как Jasmine или Versant. По мере того, как объектно-реляционные продукты достигают совершенства, несоответствие импеданса перестает быть определяющим критерием, так как при обоих подходах теперь будет поддерживаться быстрая разработка приложения. Основными характеристиками будут производительность и доступность инструментальных средств сторонних производителей.

Автор прогнозирует, что через некоторое время приложения обработки данных на основе записей должны переориентироваться на объектно-реляционные решения. Основной причиной этого является рациональное использование средств, уже затраченных на разработку программного кода и обучение персонала. Вопрос стратегий перехода организаций на новые платформы был подробно рассмотрен в главе 4.

Поставщики некоторых реляционных продуктов действенно отреагировали на критику в отношении производительности, реализовав свои продукты на машинах с массовым параллелизмом, но аналогичные действия могут быть предприняты и для повышения производительности объектно-ориентированных продуктов. Тем более что объектная модель наиболее пригодна для распределенной и параллельной реализации.

Будущие покупатели продуктов баз данных должны проверять, какие стандарты поддерживаются в рассматриваемых продуктах. Надежными стандартами в этой области являются CORBA группы ODMG, MOF и стандарты компонентов, стандарты MTS, COM+ и ODBC фирмы Microsoft, стандарт EJB корпорации Sun и стандарты языков OQL и SQL3.

Если уважаемый читатель внимательно изучил предыдущие главы, то он понимает, что объектно-ориентированные языки программирования все еще разрабатываются, а объектно-ориентированные базы данных работают довольно медленно. Автор надеется, что он также продемонстрировал и передал свой энтузиазм в отношении этой необычайно перспективной технологии. Однако для успешного использования этой технологии требуются новые методы проектирования, анализа, разработки технических требований и управления. Эти вопросы будут рассматриваться далее.

5.12. Резюме

Реляционная модель была первой формальной моделью данных. Реляционные базы данных, основанные на этой модели, были в своей основе более гибкими, чем предыдущие иерархические и сетевые системы, основанные на указателях. Однако при всей своей гибкости реляционные базы данных не могут поддерживать объектную идентичность, которая являлась положительным свойством более ранних систем. Кроме того, системы, основанные на указателях, были более эффективными. Существуют противоречия и некоторый компромисс между объектной идентичностью и непроцедурными возможностями системы запросов.

Модель данных — это математический формализм, состоящий из системы обозначений для описания данных и их структуры (информации), а также набора допустимых операций, которые используются для манипулирования этими данными. После реляционной модели

появилось несколько более или менее формальных моделей. Функциональная модель является одной из наиболее успешных по теоретическим показателям, в то время как модель "сущность-связь" — одна из наиболее широко используемых в коммерческих системах.

Функциональная модель — это формальная модель, которая может использоваться для описания сетевых систем. Имеется также ряд семантических моделей и расширений реляционной модели, которые добавляют к ней семантические возможности. Наиболее значительными среди них являются различные расширенные ER-модели, которые поддерживают иерархические и сетевые структуры для подтипов, группировки и агрегации.

Объектно-ориентированные модели включают скорее поведенческие абстракции (методы), чем чисто структурные абстракции моделей данных. Они редко поддерживают конструкторы разнообразных семантических моделей данных. Методы наследования, связанные с этими двумя подходами, также различаются. Например, модели данных в системах искусственного интеллекта часто допускают наследование по умолчанию на уровне класса, в то время как объектно-ориентированные модели позволяют наследовать только имена атрибутов. Кроме того, в моделях данных наследование зачастую ограничивается только подтипами, в то время как в объектно-ориентированном программировании допускается наследование методов.

В объектно-ориентированные базы данных и объектно-ориентированные методы необходимо привнести средства структурного моделирования, созданные разработчиками моделей данных, наряду с некоторыми идеями из области искусственного интеллекта. Вероятно, будет полезной интеграция объектно-ориентированного подхода с семантическим моделированием данных и моделями искусственного интеллекта.

В реляционных базах данных существуют трудности при работе с рекурсивными запросами, неопределенными значениями, абстрактными типами данных, а также при представлении данных и функциональной семантики. В них также отсутствует реальная поддержка бизнес-правил и правил целостности.

Нормализация скрывает и разрушает семантику, а также функциональные зависимости между атрибутами. Нормализованные отношения почти никогда не соответствуют какому-либо объекту в реальном мире. Это означает, что нормализация устраняет возможность обратного проектирования (reverse engineering). Процесс нормализации необратим.

Было предпринято несколько попыток создания реальных систем баз данных на основе полных семантических моделей данных.

Базы данных, основанные на ER-модели, и дедуктивные базы данных являются полезными гибридами, способными к решению многих проблем баз данных, для которых реляционные системы непригодны. Дедуктивные базы данных особенно полезны для создания прототипов на основе спецификаций (specification prototyping).

Появившаяся недавно объектная модель объединяет многие из возможностей и преимуществ расширенных реляционных систем, дедуктивных систем и систем, основанных на ER-моделях, а также систем, основанных на указателях. Объектная модель, по мнению автора, допускает объединение идей из области семантического моделирования данных, искусственного интеллекта и объектно-ориентированного программирования, проектирования и анализа.

Теперь разработчики стоят перед выбором между чисто объектно-ориентированными и объектно-реляционными системами.

Система управления базами данных поддерживает:

- перманентность объектов;
- управление очень большими объемами данных;

- целостность данных и транзакций;
- совместное использование и параллельный многопользовательский доступ;
- языки для доступа или запросов;
- восстановление;
- безопасность.

Объектно-ориентированная система управления базами данных объединяет возможности баз данных с возможностями объектно-ориентированного программирования, а именно обеспечивает инкапсуляцию, наследование и объектную идентичность. Можно сказать, что справедлива следующая формула.

Объектно-ориентированная база данных =
= База данных + Объектно-ориентированный подход

Объектно-ориентированные базы данных объединяют некоторые возможности семантических моделей данных и объектно-ориентированного программирования. Можно добавить также возможности экспертных систем, но этого чаще всего не делается. В них реализованы, по крайней мере, два вида ортогональных, но взаимодействующих структур: иерархии классификации и агрегации.

В ООБД устранено несоответствие импеданса между языком приложения и языком запросов. В отличие от реляционных систем, в них устранена необходимость выполнения ресурсоемких операций объединения при использовании объектов в приложении. Это делает их более эффективными для приложений, включающих сложные объекты. ООБД, наряду с высокой эффективностью, сохраняют гибкость реляционных систем. К ним добавлена поддержка для длинных транзакций и автоматического контроля версий. В некоторых из них предлагается динамическая эволюция схемы, а также поддержка средств мультимедиа и групповой работы.

В ООБД имеется ряд других преимуществ по сравнению с РСУБД, но есть и нерешенные проблемы, касающиеся непроцедурных языков запросов, оптимизации запросов и блокирования.

Известно несколько коммерческих продуктов чисто объектно-ориентированных баз данных. Они обычно применяются в приложениях, где преобладают сложные объекты, такие как Web-серверы, мультимедийные базы данных, геоинформационные системы и CAD/CAM-системы. В настоящее время пользователи, в основном, стоят перед выбором между этими продуктами и объектно-реляционными гибридами. Основным критерием выбора должна быть производительность.

Помимо чисто объектно-ориентированных баз данных, можно воспользоваться преимуществами семантических, дедуктивных или объектно-реляционных баз данных. Последний вариант предполагает реализацию объектно-ориентированного подхода посредством языка SQL3, который устраняет несоответствие импеданса, но не может решить проблему производительности. Это связано с тем, что основными приложениями объектно-ориентированных баз данных являются системы, в которых необходимы выполнение множества небольших объединений для восстановления сложных объектов, а также поддержка распределенных вычислений, контроля версий, различных платформ и коллективной работы.

Объектно-ориентированные базы данных лучше использовать в коммерческих приложениях, для которых критична производительность, в то время как объектно-реляционные системы более приемлемы в приложениях, ориентированных на записи, а также на расширение возможностей существующих систем.

Управление объектно-ориентированными базами данных — это стремительно развивающаяся технология, которая имеет большое значение для информационных технологий вообще.

5.13. Дополнительная литература

Учебник Ульмана [764] по теории баз данных стал первым подробным учебным пособием, содержащим анализ реляционной модели, но вскоре уступил пальму первенства по популярности книге [214]. Более поздняя двухтомная работа Ульмана [765, 766] включает вопросы управления знаниями и описание объектно-ориентированного подхода, а также дает хорошее представление о концепции отношений. Полное введение во все аспекты теории баз данных представляет книга [255], которая также описывает объектно-ориентированные базы данных. Все эти книги полностью охватывают теорию нормализации.

Семантические модели рассматриваются в работах [214, 255, 765, 766]. Некоторые важные конструктивные статьи по моделям данных собраны в [829].

Публикации [399, 620] — превосходные обзоры методов семантического моделирования данных, содержащие сравнение с объектно-ориентированными моделями и принципами искусственного интеллекта. Работа [754] является превосходным обзором ER-моделей и представляет одну важную расширенную версию и связанный с нею метод проектирования.

В [450] рассматриваются различия между семантическими моделями и объектно-ориентированным подходом. Работы [116, 117] охватывают различные проблемы, связанные с семантическими, интеллектуальными и дедуктивными базами данных, а также с объединением представлений из теории баз данных, искусственного интеллекта и объектно-ориентированного программирования.

Работа [730] описывает суть проектных решений, связанных с расширением реляционной базы данных интеллектуальными и объектно-ориентированными возможностями.

Связи между семантическими моделями данных и объектно-ориентированным подходом детально и глубоко рассматриваются в [341]. Эта работа представляет собой лучшее введение в семантические модели.

В [155] достаточно подробно исследованы основные коммерческие продукты ООСУБД. Объектно-ориентированным базам данных посвящены книги [347, 473]. В [341] представлены два тематических раздела. В первом приводится краткий обзор современной теории и практики баз данных, подчеркивающий необходимость включения в объектно-ориентированные базы данных идей семантического моделирования данных и искусственного интеллекта. В этом разделе дается превосходный критический анализ объектно-ориентированных баз данных и строгих традиций объектно-ориентированного программирования. Второй тематический раздел — это отчет группы авторов по довольно специализированному исследованию в области экспериментальных языков и систем баз данных, а также созданию приложения с базой данных из предметной области биологии. Комментарий по этому разделу содержит важные соображения о возможности создания унифицированных приложений. Эта книга представляет существенный интерес для тех, кто серьезно рассматривает вопрос использования объектно-ориентированных баз данных. Книга [148] охватывает расширенные

реляционные продукты. В [12] приводится сравнительный анализ шести программных продуктов с точки зрения инженерно-технических приложений.

Книга [449] является введением в основные возможности системы ORION. Она представляет собой сборник статей, а также включает материалы по GemStone и прототипу объектно-ориентированной базы данных OZ+, предназначенному для приложений автоматизации делопроизводства. Книга [447] — это общее руководство по объектно-ориентированным базам данных, которое почти полностью сконцентрировано на примерах системы ORION и может рассматриваться в качестве первоклассного введения в принципы, заложенные в систему Itasca. Еще одним превосходным исследованием принципов, на сей раз в контексте O_2, является работа [222].

В [795] дан превосходный обзор объектно-ориентированных баз данных. Работа [441] предоставляет более глубокие исследования, включая рассмотрение проблем архитектуры, которые не были включены в эту книгу.

Система Vbase описана в работах [35, 255], которые также являются великолепными учебниками по базам данных вообще. Еще один превосходный учебник по современной теории баз данных [765, 766] подробно описывает язык OPAL системы GemStone.

Перманентные языки программирования рассмотрены в работе [43], в то время как работа [667] дает краткое представление об исследованиях в этой области. Важный перманентный язык программирования Galileo описан в [16].

Работа [829] содержит представительную выборку оригинальных документов по объектно-ориентированным базам данных, хотя материал по важному вопросу эволюции схемы опущен. Редакторы включили в книгу обширный вводный информационный материал. Интересные данные, касающиеся эволюции схемы, можно найти в статьях по системе ORION [449, 589], которые представляют некоторые методики для автоматического распространения изменений и динамической классификации экземпляров. Другие материалы исследований можно найти в работе [541. Книга [142] является собранием конструктивных статей на тему интеграции объектно-ориентированных и семантических моделей данных.

Книга [829] содержит информацию из различных областей вычислительной техники. Она включает оригинальные статьи по своей тематике и, наряду с превосходным журналом *Computing Surveys* ассоциации по вычислительной технике ACM (Association for Computing Machinery), является хорошим способом расширения кругозора по современной вычислительной технике без какого-либо риска вульгаризации.

Среди нескольких обзоров работ по интегрированию систем баз знаний и данных достойны упоминания работы [116, 117, 765, 766].

Книга [618] является введением в большинство родственных технологий и описывает конкретную модель, разработанную авторами.

Журнал по объектно-ориентированному программированию JOOP (Journal of Object Oriented Programming) является хорошим источником информации о современных тематических исследованиях, написанной на довольно высоком уровне. Информация о новейших программных продуктах различных производителей содержится на соответствующих Web-узлах.

5.14. Упражнения

1. Как в объектно-ориентированных базах данных называется существование объектов после завершения работы прикладной программы?
   а) статическое связывание
   б) перманентность
   в) двухфазное завершение
   г) объектная целостность
2. Что такое несоответствие импеданса?
3. Каким образом в реляционной базе данных может быть реализовано наследование?
4. В чем на практике заключается основное отличие объектно-ориентированной базы данных от объектно-реляционной? Укажите области применения для каждого из подходов.
5. Подробно опишите известную вам объектно-ориентированную или объектно-реляционную базу данных в контексте известного приложения.
6. Рассмотрите проблему блокирования в базах данных, поддерживающих наследование. Каково различие между оптимистическим и пессимистическим управлением параллелизмом?
7. Почему производительность чисто объектно-ориентированной базы данных может превышать производительность реляционной или объектно-реляционной базы данных?
8. Каковы преимущества и недостатки выполнения методов на сервере?
9. Что такое прозрачность размещения данных и в чем ее важность? Что такое прозрачность ссылок? Почему объектно-ориентированные системы (включающие базы данных) не обеспечивают прозрачность ссылок? Укажите их преимущества и недостатки. Обоснуйте свое мнение на примерах.

6

Объектно-ориентированный анализ и проектирование

совместно с Аланом Камероном Уилсом

Воля и высокое соизволение Небес предоставляют ему свободу в его низких замыслах, чтобы он, повторяя преступления, мог множить свои вечные муки.

Мильтон. *Потерянный Рай*

В этой главе авторы отступят от обсуждения инструментальных средств, языков и реализации и перейдут к тщательному рассмотрению методов объектно-ориентированного анализа и проектирования. Авторы сосредоточатся на рассмотрении широко используемой системы обозначений UML и на принципах моделирования. Рассмотрение, в частности, базируется на методах Catalysis [204] и SOMA [327]. Обращается внимание на лучшие практические примеры и предлагается набор требований для практической методики анализа и проектирования.

Эта и следующие три главы являются основными главами этой книги. Их цель состоит в том, чтобы объединить несколько тем, которые изложены в других главах. В этой главе вопросы объектно-ориентированного программирования и его преимуществ, вопросы экспертных систем и инженерии знаний, а также вопросы моделирования данных и управления базами данных объединены в ракурсе требований для практической, строгой, объектно-ориентированной разработки программного обеспечения.

Стратегическая проблема, поставленная развивающимися технологиями объектно-ориентированного программирования, баз данных, промежуточного или связующего программного обеспечения (middleware) и компонентной разработки, решается с помощью подхода к анализу и проектированию. Данный подход должен обеспечить поэтапное, постепенное использование преимуществ объектно-ориентированных методов на уровне анализа и, соответственно, заложить прочные основы для применения языка и объектно-ориентированных технологий управления объектами. В главе 7 мы снова вернемся к объектно-ориентированному проектированию в контексте компонентно-ориентированной разработки, а также рассмотрим важные проблемы архитектуры и шаблонов. В главе 8 продолжим анализ методов спецификации программного обеспечения, а также рассмотрим вопросы инженерии требований и моделирования бизнес-процессов. В главе 9 обсудим вопросы управления и жизненного цикла программного продукта, возникающие при использовании объектно-ориентированных и компонентно-ориентированных методов разработки.

Авторы представляют и описывают унифицированный язык моделирования UML (Unified Modelling Language). Язык UML это стандартизированная система обозначений для объектно-ориентированного анализа и проектирования. Однако метод — это больше чем система обозначений. Для того чтобы стать методом анализа или проектирования, язык должен быть дополнен руководящими принципами для использования системы обозначений и методологическими основами их применения. Законченный метод разработки программного обеспечения должен также включать процедуры для решения вопросов, выходящих за рамки простой разработки программного обеспечения. К ним относятся моделирование предметной области и требований, процесс разработки, управление проектом, метрики, методы оперативного контроля и управления возможностью повторного использования. В этой главе основное внимание уделяется аспектам, касающимся обозначений, анализа и проектирования.

6.1. История развития объектно-ориентированных методов анализа и проектирования

Развитие теории вычислительных систем в самом начале касалось исключительно программирования, и только недавно это затронуло методы анализа и проектирования. И возможно, как отражение этой тенденции, интерес к объектно-ориентированному подходу также исторически начался с развития языков. Объектно-ориентированные методы проектирования возникли только в 1980-х годах. В 1990-е годы появились объектно-ориентированные методы анализа.

Не считая нескольких малоизвестных ИИ-приложений, вплоть до 1980-х годов объектная ориентация была в значительной степени связана с разработкой графических интерфейсов пользователя GUI (Graphical User Interface). Приобрели широкую известность несколько других приложений. До этого времени совсем не упоминалось об анализе или проектировании объектно-ориентированных систем. В 1980-х годах Гради Буч (Grady Booch) опубликовал статью под пророческим заголовком: *Object-Oriented Design* (Объектно-ориентированное проектирование), в которой были изложены вопросы проектирования для языка Ада. К 1991 году Буч смог расширить свои идеи до подлинно объектно-ориентированного метода проектирования, который он описал в своей одноименной книге, переработанной в 1993 году [99].

1990-е годы характеризовались как возросшей напряженностью в коммерческой деятельности, так и доступностью более дешевых и намного более мощных компьютеров. Это привело к повышению спроса и к разработке приложений, выходящих за рамки GUI-интерфейсов и ИИ. Распределенные, открытые вычисления стали не только возможными, но и важными, и объектная технология легла в основу большей части проектов по разработке программных систем, особенно с появлением n-уровневых систем клиент/сервер и сети Web, хотя реляционные базы данных играли и продолжают играть важную роль. Появление новых приложений и усовершенствованных аппаратных средств означало, что основные организации приняли объектно-ориентированное программирование и теперь хотели бы уделить соответствующее внимание объектно-ориентированному проектированию и (последующему) анализу, интерес к которому проявился с 1990-х. Интерес к объектно-ориентированному подходу в выработке требований (к проектируемой системе) наблюдался много позже.

Первая книга под названием "Object-Oriented: Systems Analysis" (Объектно-ориентированный: анализ систем) была написана Шлеер (Shlaer) и Меллором (Mellor) в 1988 году [707]. Как и в первой работе Буча, в ней не был представлен объектно-ориентированный метод в чистом виде: она была полностью сконцентрирована на описании расширенных моделей "сущность-связь", основанных, по существу, на реляционном представлении проблемы и игнорировании поведенческих аспектов объектов. В 1992 году Шлеер (Shlaer) и Меллор (Mellor) издали второй том, в котором утверждалось, что поведение должно моделироваться с использованием традиционной диаграммы состояний, а также была заложена основа истинной объектной ориентации. Однако в работе не приуменьшалась роль подхода, управляемого данными. Он остается важным благодаря его идее "трансляционного" моделирования ("translational" modelling), которая будет обсуждаться ниже. Между тем Питер Коад (Peter Coad) ввел поведенческие идеи в простой, но объектно-ориентированный метод [171, 172]. Метод Коада сразу же стал популярным благодаря своей простоте, а метод Буча — благодаря своей прямой и всесторонней поддержке возможностей языка C++, наиболее популярного в то время объектно-ориентированного языка программирования в коммерческом мире. Это сопровождалось взрывом интереса и ростом количества публикаций на тему объектно-ориентированного анализа и проектирования. Не считая тех, которые уже были упомянуты, среди наиболее значительных были метод OMT [521, 674], OOSE [417] и ROD [803]. Метод OMT стал еще одним укоренившимся методом, основанным на данных, поскольку он применялся при проектировании реляционных баз данных. Он быстро стал доминирующим, поскольку большинство программистов в то время были вынуждены заниматься написанием на языке C++ программ, взаимодействующих с реляционными базами данных.

Метод OMT [674] подобен методу Коада, в котором к описаниям типа сущности добавляются операции для создания моделей классов, но метод OMT использует отличную от всех предыдущих методов систему обозначений. Метод OMT не только полностью основан на данных — в нем процессы отделены от данных за счет разделения диаграмм потоков данных и классов. Это еще раз подчеркивает важную роль диаграмм состояний для описания жизненного цикла экземпляров. Также было сделано несколько замечаний относительно процесса микроразработок и был предложен очень полезный совет, касающийся объединения объектно-ориентированных программ с реляционными базами данных. Точно так же, как Буч (Booch) стал популярен среди программистов на C++ из-за его способности точного моделирования семантических конструкций этого языка, так и OMT стал популярен среди разработчиков, для которых язык C++ и реляционная база данных были основными инструментальными средствами.

В работе [82] это подтверждается следующими словами: "объектная модель OMT является, по существу, расширением подхода сущность-связь" (с. 10). Они, представляя версию OMT второго поколения, продолжают утверждать, что "авторы UML обращаются к программным приложениям; мы же обращаемся к приложениям баз данных". В предисловии к этой же книге Румбах рассматривает реляционный характер OMT в качестве его преимущества. Мне кажется, что более четкое следование объектно-ориентированным принципам и подходу, основанному на обязанностях (responsibility-driven approach), необходимо в том случае, если все преимущества объектно-ориентированного модельного представления могут быть получены в среде полностью объектно-ориентированных инструментальных средств.

Наряду с появлением расширенной модели сущность-связь и методов, основанных на управлении данными (data-driven methods), Вирфс-Брок (Wirfs-Brock) и ее коллеги разрабатывали набор методов проектирования, основанных на обязанностях RDD (Responsibility-Driven Design). Они основывались на опыте, приобретенном больше в среде Smalltalk, чем в среде реляционных баз данных. Наиболее важным вкладом RDD было расширение идеи использования так называемых CRC-карт для проектирования и позднее для представления идеи стереотипов. CRC-карты позволяют представить классы (Class), их обязанности (Responsibility) и взаимодействие (Collaboration) с другими объектами и являются отправной точкой для проектирования. На семинарах по проектированию они могут быть "перетасованы", а обязанности перераспределены. Такой подход возник на основе работы Бека (Beck) и Каннингхема (Cunningham) [64] из фирмы Tektronix, где карты были реализованы с использованием гипертекстовой системы. Перемещение на физические части платы расширило методику, что позволило проектировщикам наделять свои классы человеческими качествами и даже рассматривать представление их жизненных циклов. Эти идеи интенсивно используются в главе 9.

Метод Objectory — это патентованный метод, который был популярен намного дольше, чем большинство объектно-ориентированных методов. Он возник в шведской телекоммуникационной промышленности и появился в объектно-ориентированном виде, когда Якобсон и его коллеги опубликовали часть этого метода (OOSE) в виде книги [417]. Основным вкладом этого метода была идея о том, что классификации должен предшествовать анализ прецедентов (случаев использования). Тогда классы создаются на основе прецедентов. Этот метод обеспечил важный шаг вперед в объектно-ориентированном анализе и был широко принят, несмотря на то что в его адрес были направлены строгие критические отзывы. Метод Objectory был первым ОО-методом, который включал *подлинный* (*bona fide*), хотя и частичный процесс разработки.

Анализ поведения объектов OBA (Object Behaviour Analysis) возник на основе опыта работы преимущественно с системой Smalltalk в компании ParcPlace и также включает модель процесса, которая полностью никогда не была опубликована, хотя некоторая информация и была доступна [304, 672]. Одной из интересных особенностей анализа OBA было использование стереотипных сценариев вместо прецедентов.

Исходя из традиции языка Eiffel, Уолден и Нерсон [772] в своей системе обозначений для бизнес-объектов BON (Business Object Notation) подчеркнули удобство для пользователя и важность процесса прототипирования. Однако этот подход (и его значительное удобство для пользователя) зависел от принятия языка Eiffel в качестве языка спецификации на всем протяжении процесса. Этот подход также внес важный вклад в строгость объектно-ориентированного анализа, как и [187]. В подходе BON строгость повышается посредством использования понятия инвариантов класса в языке Eiffel, в то время как при подходе Syntropy это реализуется с помощью конечных автоматов.

Метод MOSES [378] стал первым ОО-методом, включающим полный процесс разработки, набор метрик и подход к управлению повторным использованием. Метод SOMA [327], который появился в сформированном виде почти одновременно с методом MOSES и находился под его влиянием, включает все эти возможности, а также пытается объединить самые лучшие стороны всех опубликованных до настоящего времени методов и пойти дальше, особенно в области разработки технических условий, процесса, агентских систем и строгости.

Во втором издании этой книги в 1994 году автор насчитал свыше 72 методов или их фрагментов. Многие из них несколько подробнее рассматриваются в приложении Б. Специалисты по объектно-ориентированному (ОО) подходу скоро поняли, что такая ситуация непригодна для промышленного использования этой технологии в больших масштабах. Они также осознали, что большинство методов в значительной степени перекрываются. Поэтому имели место различные инициативы, направленные на объединение и стандартизацию методов. До сих пор перед разработчиком был настоящий "коктейль" из объектно-ориентированного анализа, методов проектирования и систем обозначений. Явным событием была попытка некоторой унификации и появление обобщающего метода "слияния" Fusion [179, 515]. Он представляет одну из первых попыток объединения оптимальных методов, хотя некоторые оценили эту коллекцию методов как недостаточно интегрированную. Работы по "слиянию" методов продолжаются, хотя публикации по ним появляются в незаконченном виде, по сравнению с патентованными версиями, продаваемыми корпорацией Hewlett-Packard. Разработчик в области современной ОО-системы должен найти способ выбрать из этого "коктейля" наиболее пригодный для себя метод. По этой причине и из-за сходства методов большинство специалистов по методологии осознали, что необходимо начать процесс объединения этих методов.

Консорциум OPEN являлся неофициальной группой, состоящей приблизительно из 30 специалистов по методологии, без обычного коммерческого членства. Эти специалисты добивались большей интеграции методов, но были убеждены в том, что методы должны включать полный процесс, должны быть всеобщим достоянием, не должны быть привязаны к конкретным инструментальным средствам и должны быть сфокусированы строго на научной целостности, а также на практических проблемах. Основателями консорциума OPEN были Брайен Хендерсон-Селлерс (Brian Henderson-Sellers) и непосредственно сам автор, которые начали интегрировать модели процесса MOSES и SOMA. Результат был опубликован в работе [334]. Вскоре к ним присоединился Дон Файерсмит (Don Firesmith). Он начал работать над интегрированной системой обозначений (OML) с целью демонстрации более строгой объектно-ориентированной природы обозначений, чем в языке UML, находящемся под влиянием метода OMT. Эта система обозначений должна стать более легкой для изучения и запоминания [272].

Джим Румбах оставил компанию GE, чтобы присоединиться к Гради Бучу (Grady Booch) из корпорации Rational Inc. Они объединили свои системы обозначений. Так появилась первая версия языка UML [100]. Позже к ним присоединился Айвар Якобсон (Ivar Jacobson), который добавил элементы своей системы обозначений Objectory и начал разработку унифицированного процесса RUP (Rational Unified Process) (см. главу 9). Язык UML был представлен на рассмотрение группе OMG для принятия в качестве стандарта, а множество других специалистов по методологии вынесли на рассмотрение свои идеи, хотя поставщики инструментальных средств, разработанных на основе CASE-технологии, в большинстве случаев сопротивлялись как нововведениям, которые заставили бы их переписать свои инструментальные средства, так и упрощениям, которые могли сделать их инструментальные средства менее полезными. Консорциум OPEN предложил семантически более богатый язык OML, которым большей частью пренебрегали, несмотря на многие хорошие идеи; вероятно, в

большинстве случаев из-за его чрезмерной сложности [272]. В язык были добавлены элементы реального времени (real-time elements), базирующиеся на методе ROOM [694], и, кроме того, там был представлен формальный язык ограничений OCL, созданный под сильным влиянием Syntropy [188]. Система обозначений для множества интерфейсов к классам базировалась на работах по модели COM+ фирмы Microsoft. Диаграммы видов деятельности для моделирования процесса были построены с использованием метода Мартина—Оделла (Martin-Odell). Концепция стереотипов, принятая в языке UML, была основана на идеях, предложенных Ребеккой Вирфс-Брок (Rebecca Wirfs-Brock) (хотя и сильно искаженных в первых реализациях стандарта). Борьба за улучшение языка UML продолжается, и поэтому в этой книге не предполагается рассматривать обозначения, полностью принятые в этом языке. Таким образом, проблемы с системой обозначений в значительной степени были разрешены к концу 1990-х, когда акцент переместился на новшества в области методов и процессов. Наиболее значительный вклад в анализ и методологию проектирования, после создания языка UML, внес метод Catalysis [204], который был первым методом, содержащим конкретные технологии компонентной разработки наряду с логически последовательным руководством по использованию UML. Собственный опыт автора показал, что объекты могут рассматриваться в качестве интеллектуальных агентов, если к спецификации типа будут добавлены наборы правил. Это вызвало настоятельную необходимость инвариантов для полного определения типов в других методах (особенно BON, Syntropy и Catalysis).

На рис. 6.1 показаны связи между отдельными объектно-ориентированными методами, языками и системами обозначений, обсуждаемыми в этой книге. Для рассмотрения этих и других методов можно обратиться к приложению Б.

6.2. Инженерия программного обеспечения

Объектно-ориентированные методы включают, по меньшей мере, методы проектирования и анализа. В некоторых случаях эти методы перекрываются, а в действительности только для идеальных случаев можно говорить, что они являются полностью независимыми процессами. В [389] утверждается, что процесс разработки систем — это процесс осмысления, изобретения и реализации, в соответствии с которым предметная область осмысливается и постигается через феномен, понятия, сущности, действия, роли и утверждения. Это осмысление целиком и полностью соответствует анализу. Однако понимание предметной области также влечет за собой одновременно осмысление структуры, компонентов, вычислительных моделей и других интеллектуальных построений, которые учитываются в области допустимых решений. Эта изобретательская деятельность соответствует процессу проектирования. Конечно, наиболее традиционные философы в области проектирования программного обеспечения ужаснутся, узнав предположение автора о том, что понимание ответов предшествует, до некоторой степени, пониманию проблемы. Подобным образом протекают все другие когнитивные процессы, и автор не видит причин, по которым проектирование программного обеспечения должно от них отличаться. Эти соображения также относятся к процессу реализации, где эти структуры и архитектурные компоненты проецируются на компиляторы и аппаратные средства. Защитники эволюционной разработки долгое время утверждали, что выгодно не проводить жесткого разделения между анализом, проектированием и реализацией. С другой стороны, организационные и исполнительские соображения ведут к серьезным вопросам о целесообразности создания прототипа в промышленной среде. В [315] предложен ряд способов

управляемого создания прототипов. Эта идея подробнее рассматривается в главе 9. В корне этих дебатов находятся онтологическая и эпистемологическая позиции относительно того, что такое объекты и как их можно осмыслить или понять. Эти вопросы также рассматриваются в этой главе.

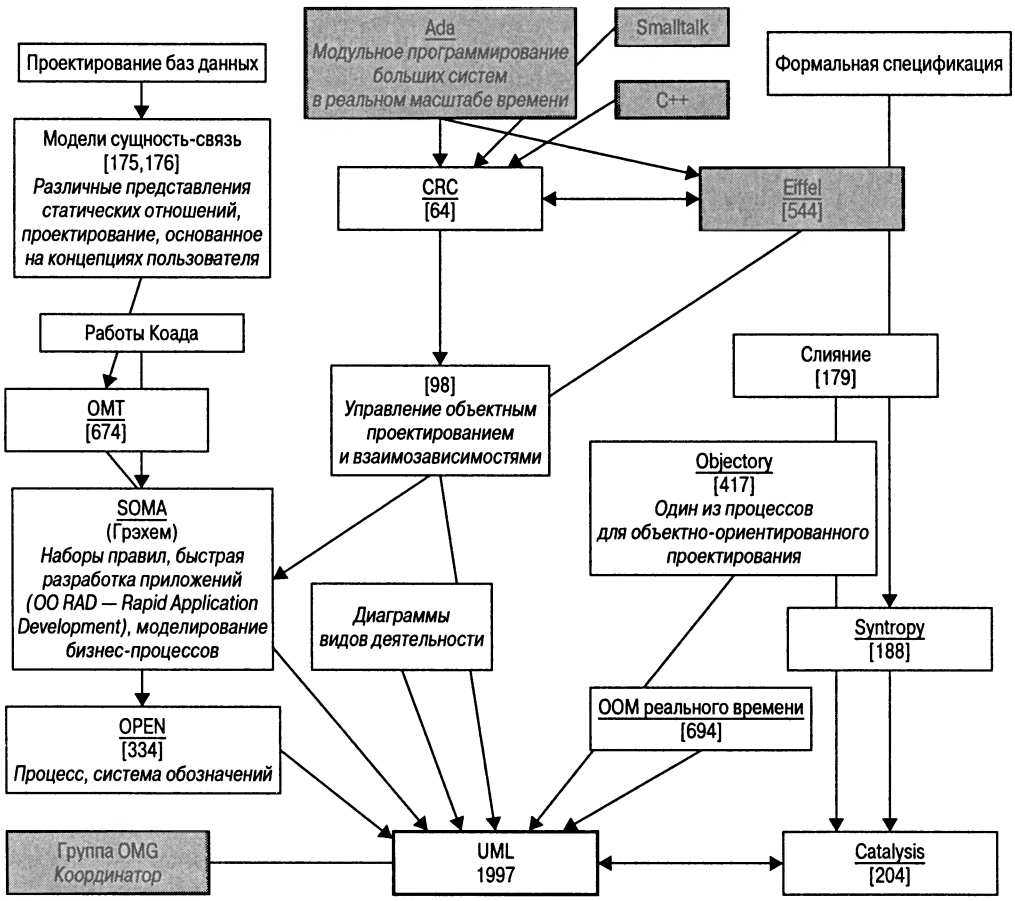


Рис. 6.1. Некоторые факторы, определившие появление языка UML

Спецификации возможностей повторного использования

В [76] отмечено, что из повторно используемых компонентов может быть построено меньше половины типичной программной системы и что единственным способом получения значительного роста производительности и качества является повышение уровня абстракции компонентов. Анализ программных продуктов или спецификаций является более абстрактным процессом, чем проектирование. Проектирование более абстрактно, чем кодирование. Абстрактные артефакты (artefact) менее детализированы и в меньшей степени зависят от аппаратных средств и других ограничений реализации. Таким образом, преимущества

возможности повторного использования могут быть получены на стадии проектирования, когда они, вероятно, будут иметь самое большое влияние. Но менее детализированный объект в меньшей степени поддается интерпретации. В случаях, когда важны расширяемость или семантическое богатство языка, может потребоваться большая детализация, что, до некоторой степени, может усложнить повторное использование. Это заставляет задуматься, существуют ли объектно-ориентированный анализ и методы проектирования, которые могут предоставить преимущества повторного использования и расширяемости. В условиях активного развития объектно-ориентированного программирования и компонентной технологии вопрос о том, можно ли получить эти преимущества прямо сейчас, до появления более развитых и устойчивых языков и структур, приобретает еще большую значимость. Автор полагает, что можно. Однако на уточняющий вопрос о том, какие методы проектирования и анализа необходимо использовать, ответить сложнее. Популярной системой обозначений является язык UML, который первым был принят в качестве стандарта группой OMG в 1997 году, но язык UML — это только система обозначений. Чтобы превратить его в метод, к нему необходимо добавить методику и руководящие принципы.

Фирмы по разработке или поставке программного обеспечения, а также консалтинговые компании должны быть особенно заинтересованы в расширяемых спецификациях, которые можно повторно использовать. Для этого есть финансовая причина. Сотрудники фирм по поставке программного обеспечения и консалтинговых компаний работают с клиентами, чтобы понять их бизнес и требования, а затем помочь им изготовить программное обеспечение для решения их проблем. Приобретя ценный опыт, консультанты затем продолжают работу со следующим клиентом и продают ему то, чему они научились, возможно, за более высокую плату, оправданную дополнительным знанием. Некоторые фирмы идут дальше. Они пытаются вложить свой опыт в настраиваемые функциональные спецификации. Например, фирмой BIS Information Systems, в которой работал автор, в 1980-е годы был разработан программный продукт, называемый "моделью ипотеки", который являлся функциональной спецификацией ипотечного приложения, базирующегося на ряде аналогичных проектов. Его можно было приспособить к потребностям конкретного клиента. Трудность заключалась в том, по крайней мере для фирмы BIS, что модель ипотеки не могла быть продана продавцам фруктов и овощей или фирмам-изготовителям стиральных машин, даже при том, что некоторые из сущностей, такие как счета, могли быть применены ко всем этим коммерческим предприятиям. В этом случае требуется набор многократно используемых компонентов спецификации, которые могут быть собраны в функциональную спецификацию, пригодную для *любого* коммерческого предприятия. Объектно-ориентированный анализ и, в меньшей степени, проектирование обещают предоставить такую возможность, даже если единственные, существующие в настоящее время каркасы с возможностью повторного использования, такие как San Francisco фирмы IBM, все еще поставляются в виде кода.

Для определения терминологии необходимо начинать со значительно упрощенного изображения процесса разработки программного обеспечения или его жизненного цикла. Согласно этой упрощенной модели, разработка начинается с установления требований предметной области и сбора инофрмации о ней, а заканчивается тестированием и последующим сопровождением. Между этим происходит три главных действия: определение технических требований и логическое моделирование (анализ), архитектурное моделирование (проектирование) и реализация (кодирование и тестирование). Конечно, эта модель допускает итерации, создание прототипов и другие отступления, но они на этой стадии не будут рассматриваться. Итерационная и эволюционная разработки наряду с организационными проблемами, возникающими при этом, подробно рассматриваются в главе 9. В реальной

жизни, вопреки учебникам, процессы определения технических требований и проектирования в значительной степени перекрываются. Это особенно верно для объектно-ориентированного проектирования и анализа, поскольку абстракции для них моделируются скорее на основе абстракций приложения, а не процессоров и дисков. Как известно, проектирование может быть разделено на логическое и физическое. При объектно-ориентированном проектировании логическая стадия часто неотличима от некоторых частей объектно-ориентированного анализа. Одной из главных проблем, с которыми сталкиваются при структурном анализе и методах проектирования, является отсутствие пересечения или плавного перехода между двумя стадиями. Это часто приводит к трудностям в отслеживании продуктов проектирования в обратном направлении к первоначальным требованиям пользователя или продуктам анализа. Подход, принятый в объектно-ориентированном анализе и проектировании, ведет к объединению системного анализа с процессом логического проектирования, несмотря на то что все еще имеется различие между выявлением требований и анализом, а также между логическим и физическим проектированием. Тем не менее объектно-ориентированные анализ, проектирование и даже программирование, благодаря их применению в течение всего жизненного цикла разработки в соответствии с унифицированной концептуальной моделью объектов, по крайней мере, позволяют преодолеть некоторые проблемы отслеживания, связанные с разработкой систем. Одной из главных причин этого является разнообразие форм представления, которые разработчик объектно-ориентированного программного обеспечения использует при выполнении анализа, проектирования и программирования. Во всех этих переходах используется, по сути, одна и та же единица измерения — объект. Аналитики, проектировщики и программисты могут использовать одно и то же представление, одну и ту же систему обозначений и одно модельное представление, а не диаграмму потоков данных DFD (Data Flow Diagram) на одном этапе, структурные диаграммы на другом и т.д. Преимущества объектно-ориентированного анализа и проектирования, которые, возможно, очевидны для читателя, ознакомившегося с замечаниями в предыдущих главах, сводятся к следующему.

- Необходимые изменения локализуются, а непредвиденные взаимодействия с другими программными модулями маловероятны.
- Наследование и полиморфизм делают ОО-системы более расширяемыми, способствуя, таким образом, более быстрой разработке.
- Проектное решение, основанное на концепции объектов, удобно для распределенной, параллельной или последовательной реализации.
- Объекты в большей степени соответствуют сущностям в предметной области проектировщика и пользователя, что ведет к повышению незаметности для пользователя (seamlessness) и трассируемости (traceability).
- Области совместно используемых данных инкапсулированы, что уменьшает возможность непредвиденных изменений или других аномалий модификации.

Для объектно-ориентированного анализа и объектно-ориентированных методов проектирования общими являются следующие основные этапы, хотя их порядок и детали реализации в значительной степени варьируются.

- Поиск способов, с помощью которых система взаимодействует со своим окружением (прецеденты)

- Идентификация объектов и имен их атрибутов и методов
- Определение связей между объектами
- Определение интерфейса(ов) каждого объекта, а также обработки исключительных ситуаций
- Реализация и тестирование объектов
- Компоновка и тестирование системы

Анализ — это разбиение задачи на составляющие. В обработке данных это понимается как процесс спецификации структуры и функций системы, независимо от средств реализации или физического разбиения на модули или компоненты. Анализ традиционно выполнялся сверху вниз с использованием структурного подхода или эквивалентного метода, базирующегося на функциональном разбиении и анализе данных. Часто стратегический анализ высокого уровня, управляемый производственными целями, изолируется от системного. В данном случае имеется оба типа. Это возможно, поскольку объектно-ориентированный анализ позволяет описывать систему в характеристиках реального мира; системные абстракции более или менее точно соответствуют сущностям бизнес-процесса. В главах 8 и 9 это различие вводится вновь, но с точки зрения выработки требований и управления проектом это не только необходимо, но и удобно.

Объектно-ориентированный анализ — это анализ, который содержит и элемент синтеза. Формулировка требований пользователя и идентификация ключевых объектов предметной области сопровождаются компоновкой этих объектов в структуры, которые на более поздней стадии составят основу физического проектного решения. Аспект синтеза присутствует обязательно, поскольку анализируется система; другими словами, на предметную область накладывается определенная структура. Это не значит, что проектное решение не будет уточняться. Хорошо детализированный проект может значительно отличаться от лаконичной модели спецификации.

Как видно из главы 5, имеется богатая теория семантического моделирования данных, выходящая далеко за пределы обычного использования ER-диаграмм. Эта теория затрагивает многие понятия объектной ориентации, такие как, например, наследование и абстрактные типы. Она также помогает понять суть связей или ассоциаций между сущностями. Многие результаты из этой области игнорировались специалистами в области объектной технологии и в области искусственного интеллекта с такой же тщательностью, с какой эти две области игнорировали друг друга.

Ранние методы объектно-ориентированного анализа

Часто, помимо идентичности, в системе выделяют три основных аспекта, а именно: данные, объекты или концепции и их структуры; архитектура или невременные (atemporal) процессы; и динамика или поведение системы (рис. 6.2). Эти три аспекта (или измерения) можно рассматривать в качестве данных, процесса и управления. Объектно-ориентированная методология объединяет два из этих измерений (данные и процесс), инкапсулируя локальное поведение с данными. Далее будет видно, что можно также инкапсулировать управление. Таким образом, объектно-ориентированный анализ может рассматриваться в виде силлогизма,

идущего от частного (Particular) (классов) через единичное (Individual) (экземпляры) к общему (Universal) (управлению)[1].

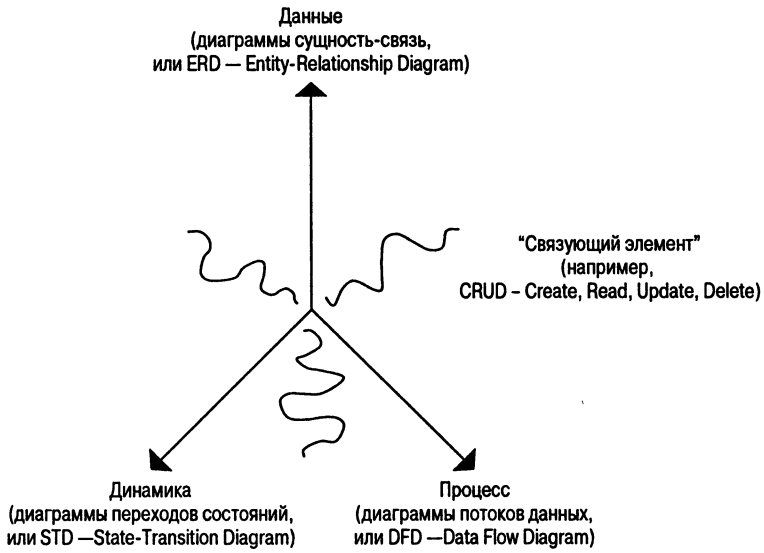


Рис. 6.2. Три измерения разработки программного обеспечения

Обыкновенный здравый смысл в разработке программного обеспечения говорит о том, что система должна описываться в соответствии с этими тремя измерениями, а именно: процессом, данными и динамикой или управлением. Измерение данных соответствует диаграммам "сущность-связь" (ERD — Entity-Relationship Diagram) или логическим моделям данных. Модели процессов представляются диаграммами потоков данных или видов деятельности. Наконец, динамика описывается или диаграммами состояний, или определением жизненного цикла сущности. При использовании структурных методов, для того чтобы гарантировать непротиворечивость этих диаграмм, обычно требуется создание каких-то документов с перекрестным контролем, которые "склеивают" модель. Например, чтобы проверить непротиворечивость модели в представлениях "сущность-связь" и диаграммы потоков данных, может быть создана матрица CRUD (эта аббревиатура означает "Create, Read, Update, Delete"). Начальные буквы указывают, какие сущности и каким образом используются процессами. Этот подход приводит к огромным непроизводительным затратам только по части документирования. Однако если предположить наличие полных знаний, то этот подход гарантирует охват всех аспектов системы. Кроме того, для получения одного и того же заключения используются две методики, и, если результаты согласуются, степень уверенности в них возрастает.

Подходы к разработке программного обеспечения на основе данных, начинаются с построения модели данных, в то время как подходы, ориентированные на процессы, начинаются с создания DFD или диаграмм видов деятельности. Некоторые подходы реального времени

---

[1] Под силлогизмом здесь понимается скорее определение, данное Аристотелем, а не общепринятый термин, значение которого сужено до чисто математического и дедуктивного умозаключения. Таким образом, силлогизм — это пример умозаключения, включающий и связывающий суждения в трех различных категориях.

начинаются с построения конечных автоматов или диаграмм переходов состояний STD (State-Transition Diagram), но это несвойственно для разработчиков коммерческих систем. Одна из проблем с диаграммами переходов состоит в том, что они прекрасно подходят для систем с небольшим числом состояний, как в случае контроллеров, но бесполезны для систем с большим числом состояний или даже непрерывными состояниями. Объект с n булевыми атрибутами может иметь $2n$ состояний. Большинство объектов коммерческих систем имеет несколько небулевых атрибутов. По этой причине необходимо сосредоточиться на состояниях, которые вызывают изменения данных, существенные для бизнес-процесса. Это означает, что нужно четко представлять и состояния, и связанные с ними изменения. Решением является разделение пространства состояний на части, соответствующие важным предикатам и точкам зрения. Например, диаграмма, составленная анестезиологом для отдельного человека, включает состояния {бодрствующий, спящий, мертвый}; составленная регистратором — {одинокий, женатый, разведенный, овдовевший, умерший}; составленная бухгалтером — {платежеспособный, неплатежеспособный}. Все эти диаграммы состояний одновременно допустимы. Но, разумеется, разделение может рассматриваться как субъективное, т.е. необходима осторожность.

При изучении ранних объектно-ориентированных методов анализа обратили на себя внимание некоторые факты. Такие методы, как метод Коада [171–173], были просты, но им недоставало возможносетй для описания динамики системы. А такие методы, как OMT [674] и метод Шлеер-Меллора (Shlaer-Mellor), обладали более широкими возможностями, но были очень сложными для изучения. Методы, подобные OMT, также оказали небольшую помощь для представления бизнес-правил и ограничений. Анализ OSA [257] и синтез IE, предложенный в [521], были немного более простыми подходами. Анализ OSA предоставлял аналитику возможность показывать ограничения на диаграммах, как своего рода размышления. Ни один из подходов не поддерживал правил, и напрасно было искать совета относительно того, как объединить результаты трех отдельных моделей в единую согласованную модель, хотя OMT в этом отношении оказался полезнее других. Попытки устранения некоторых из перечисленных недостатков в этих (во всем остальном привлекательных) методах привели автора к методу SOMA. Этот метод объединил систему обозначений для объектно-ориентированного анализа с правилами в стиле основанных на использовании знаний систем, предназначенными для описания ограничений, бизнес-правил, глобального управления системой, триггеров баз данных и определения количества связей (например, "все дети, которым нравятся похожие игрушки"). Таким образом, метод SOMA также был направлен на разрешение проблемы выработки требований (к проектируемой системе), не решенной другими методами. Это будет видно из главы 8. Кроме того, данный метод был уникален относительно поддержки нечеткого обобщения, которое является важным для разработки технических заданий в некоторых областях, таких как моделирование предприятия и управление производственным процессом, хотя он немоден среди разработчиков программного обеспечения.

По мере формирования этой дисциплины основное внимание стало уделяться моделям состояний для отдельных объектов. Современные зрелые методы освобождают разработчика от необходимости построения диаграмм потоков данных. Однако они предполагают построение моделей взаимодействия системы со своими пользователями и внешними устройствами, обычно в виде прецедентов.

6.2.1. Подходы, основанные на обязанностях и данных

Часто говорят, что данные более стабильны, чем функции, и поэтому в большинстве случаев должны быть предпочтительнее подходы, в центре которых находятся данные. Однако одним из самых больших рисков в принятии метода, базирующегося в большей степени на структурных методах, является проектное решение, управляемое данными. Два разработчика программного обеспечения в компании Boeing провели эксперимент со студентами-практикантами [699]. Одной группе преподавали метод объектно-ориентированного анализа Шлеер-Меллора (Shlaer/Mellor), основанный на данных, т.е. метод, в большей степени основанный на традиционном моделировании "сущность-связь", в то время как другая группа обучалась методикам проектирования на основе обязанностей [803]. Затем обе группы попросили разработать упрощенное приложение для управления пивоваренным заводом. Группа, изучавшая метод Шлеер-Меллора, создала проект, где большинство классов представляли статические хранилища информации, и только один класс инкапсулировал управляющие правила для большей части приложения и обращался к ним в стиле функции `main()` на языке C. Другая группа распределила поведение по всем своим классам намного равномернее. Было замечено, что при последнем подходе было создано намного больше классов многократного использования, которые могут быть использованы независимо от приложения. Это также четко продемонстрировало тот факт, что используемый метод может основательно повлиять на результат. Автор твердо убежден в том, что методы, основанные на данных, являются опасными в руках среднего разработчика и особенно в руках тех, кто воспитан на реляционных базах данных или имеет опыт работы с ними. Кроме того, автор утверждает, что огромное влияние может иметь подход, выбранный для разработки технических требований.

Исследование Шарбла и Коэна убедительно показывает, что управляемые данными методы влияют на мышление проектировщиков. Это в результате приводит к созданию классов, не пригодных для повторного использования. Обычные эффекты таковы.

- Поведение концентрируется в объектах-контроллерах, которые напоминают основные программы. Это усложняет поддержку систем из-за большого количества информации о других объектах, которую хранят эти контроллеры.
- Другие объекты имеют несколько операций и часто эквивалентны нормализованным таблицам баз данных. Это не есть согласованное объектно-ориентированное решение.

В своей производственной практике автор настаивает на использовании проектных решений и анализа, основанных на обязанностях. Это будет видно из последующих глав. Автор на протяжении всей книги подчеркивает свою приверженность основным принципам объектной технологии: инкапсуляции и наследованию. Это не педантичная реакция пуриста, а позиция огромного практического значения.

6.2.2. Трансляционный и уточняющий подходы

Еще одним важным способом, с помощью которого можно классифицировать объектно-ориентированные методы, является их разделение на трансляционные (translational) и уточняющие методы (elaborational). Методы, аналогичные методу Буча (Booch), OMT и RUP, представляют образцы уточняющих методов. В этих методах путь от спецификации до реализации рассматривается как создание начальной модели, к которой затем добавляется все

больше и больше деталей (уточнений), пока, в конечном счете, при нажатии кнопки не появится скомпилированная программа.

В трансляционных методах, среди которых метод Шлеер-Меллора (Shlaer-Mellor) является эталоном, процесс рассматривается в виде последовательности отдельных моделей и процедуры их связывания и трансляции друг в друга. Таким образом, на каждой стадии осмысления проблемы можно использовать наиболее подходящие методы моделирования и точки зрения и при этом гарантировать незаметность для пользователя (seamlessness) и трассируемость (traceability). Это будет видно позже в этой главе и в главе 7. К этому лагерю среди прочих принадлежат методы Catalysis и SOMA, и следующий раздел будет посвящен иллюстрации этого подхода.

6.3. Объектно-ориентированный анализ и проектирование с использованием UML

Унифицированный язык моделирования UML (Unified Modelling Language) — это, вероятно, наиболее широко известная и используемая система обозначений для объектно-ориентированного анализа и проектирования. Этот язык является результатом слияния нескольких предыдущих результатов в области объектно-ориентированных методов. В этом разделе он используется для иллюстрации того, как приступать к объектно-ориентированному анализу и проектированию.

Первое, что необходимо знать, — как представлять объекты. На рис. 6.3 показано представление классов и экземпляров. К сожалению, в UML не проводится адекватного различия между обозначениями для типов и классов; но для того чтобы показать различие, можно добавить стереотип `<<type>>` к пиктограмме класса. Стереотипы — это дескрипторы или теги (признаки), которые можно добавлять к объектам, чтобы классифицировать их различными способами. Эта полезная идея была первоначально предложена в [802], но в текущей версии языка UML (1.4) класс может иметь только один стереотип, что значительно снижает его значимость. Как будет видно далее, в CASE-средствах стереотип используется для графического представления объекта. Автор, как и большинство специалистов по методологии, убежден, что в будущих версиях языка UML будет разрешено иметь множественные стереотипы, как и предполагается в данной книге. Для этой цели пользователи текущих CASE-средств могут использовать комментарии (выноски) с произвольным текстом (free-text notes). Выноски с текстом также показаны на рис. 6.3. Следует обратить внимание на то, что имена экземпляров всегда подчеркиваются; иначе обозначение для экземпляра в точности совпадает с обозначением класса (типа).

Стереотип указывает вариант интерпретации элемента и способ его представления и обработки инструментальными средствами. В число стандартных стереотипов включены следующие: `<<interface>>` (интерфейс), `<<type>>` (тип) и `<<capsule>>` (капсула). Стереотипы можно добавлять к классам, ассоциациям, операциям, прецедентам (use cases), пакетам и т.д. Исполнители (actors) тоже являются объектами, отмеченными стереотипом `<<actor>>`. Большинство инструментальных средств использует стереотипы только для отображения пиктограмм (icons); например, значок "человечка". Стереотипы делают язык расширяемым, добавляя дополнительное значение к основным элементам синтаксиса, но для того чтобы сохранить связь (communicability) между моделями и изобрести что-то новое, желательно проконсультироваться с хорошим специалистом по методологии.

При объектно-ориентированном анализе сначала следует оперировать типами, и только во время проектирования целесообразно приступать к обдумыванию классов. Поэтому мы не будем использовать стереотипы. Если не обозначено иное, значок класса интерпретируется как тип.

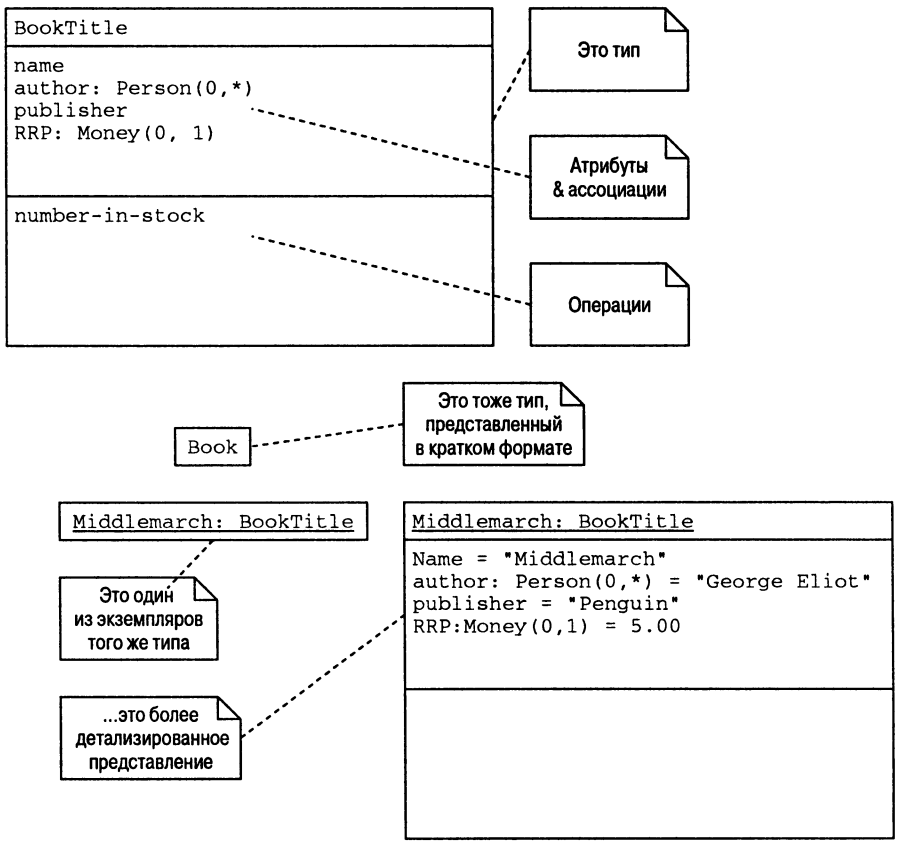


Рис. 6.3. Типы и их экземпляры

В главе 1 уже встречались некоторые обозначения языка UML. Если сравнить рис. 6.3 с рис. 1.4, *а*, то можно заметить, что при отсутствии какого-либо различия между обозначениями автор принял соглашение, что классы, которые являются коллекциями экземпляров, именуются во множественном числе, в то время как типы, которые представляют единственное понятие, — в единственном числе. Необходимо также отметить, что роли, такие, например, как роль служащего, не являются аналогами типов или классов. Причина в том, что в объектно-ориентированном программировании экземпляр принадлежит своему классу навсегда; тогда как человек может перестать быть служащим, например может уйти на пенсию.

В языке UML не имеется какого-либо адекватного обозначения для ролей, хотя примером официального обозначения является краткая форма, показанная на рис. 6.3. Роли обычно моделируются как отдельные объекты и, когда этого требует контекст, указываются со стереотипом `<<role>>`.

Напомним, что можно различать свойства класса и его экземпляров. Методы класса и атрибуты относятся к свойствам и операциям коллекций в целом. Операции, которые применяются к экземплярам, например операция вычисления возраста человека по его дате рождения, называют **операциями экземпляра** (instance operation), а операции, которые применяются к целым классам, например операция вычисления среднего возраста всех служащих, — **операциями класса** (class operation). Кроме того, имеются **атрибуты экземпляра** (instance attribute) и **атрибуты класса** (class attribute), хотя они встречаются реже.

Операции и атрибуты объекта называют его **свойствами** (feature). Они (и, возможно, имя) составляют **сигнатуру** (signature) объекта. Часто элементы определения объекта полезно рассматривать как обязанности (responsibilities). Атрибуты и ассоциации — это **обязанности знания** (responsibilities for knowing). Операции являются **обязанностями действия** (responsibilities for doing). Родовые типы (иногда называемые шаблонами) показаны на рис. 6.4.

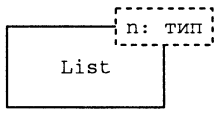


Рис. 6.4. Родовые типы или шаблоны

Одним из больших преимуществ традиционных систем управления базами данных является разделение процессов и данных, которое обеспечивает независимость данных и устойчивость к изменениям. Это реализуется благодаря стабильному интерфейсу с общими данными. Эта модель данных должна рассматриваться в качестве статической модели приложения (static model). В объектно-ориентированных системах процессы и данные интегрированы. Означает ли это, что необходимо отказаться от преимуществ, предоставляемых независимостью данных? Уже в реляционных базах данных на платформе клиент/сервер был виден шаг в направлении решения этой проблемы. Это триггеры базы данных и правила целостности, хранящиеся на сервере с данными. Поэтому при объектно-ориентированном подходе к организации данных кажется разумным принять соглашение о существовании двух видов объектов, которые автор называет **объектами предметной области** (domain object) и **объектами приложения** (application object). Объекты предметной области представляют те аспекты системы, которые являются относительно устойчивыми или общими (могут использоваться для многих приложений). Объекты приложения — это те объекты, которые предположительно могут достаточно быстро изменяться от инсталляции до инсталляции или время от времени. Этот подход возрождает идею независимости данных в расширенной, объектно-ориентированной форме. Традиционная модель данных — это представление объектов предметной области, в котором объекты также включают ограничения, правила и динамику (переходы из одного состояния в другое и т.д.). Цель состоит в том, чтобы сделать интерфейс к этой части модели максимально устойчивым. Объекты предметной области формируют общую объектную модель. Большинство взаимодействий между компонентами выполняется через эту модель. Можно пойти дальше и ввести понятие **объектов интерфейса**, единственным назначением которых является предоставление средств общения (communication) как с людьми, так и с другими системами и устройствами.

В качестве примера этого разграничения во многих предметных областях могут быть рассмотрены объекты `Products` (Изделия) и `Transactions` (Сделки), которые бесспорно

являются объектами предметной области, в то время как `DiscountCalculators` (расчеты скидок) может быть специальным объектом приложения. Классы, аналогичные `Sensors` (Датчики) и `InputValidators` (средства подтверждения правильности ввода), вероятно, будут объектами интерфейса.

Эти три категории могут рассматриваться в качестве стереотипов. К другим полезным стереотипам относятся контроллеры, координаторы, держатели информации и провайдеры услуг. Как утверждается в [802], такие классификации являются преднамеренными упрощениями, которые совершенствуются во время анализа и проектирования.

Аспекты атрибутов

В обозначении типа показывают списки ассоциаций и операций. Ассоциации являются атрибутами, которые относятся к другим типам. Есть два способа их интерпретации: их можно рассматривать или в качестве стенографической записи соответствующих им операций `get` (получить) и `set` (установить), или как словарь, с помощью которого можно изучать тип. Как было показано в [792], последняя интерпретация больше подходит для спецификации компонента или системы, поскольку ассоциации могут обеспечить базис для части средств тестирования. Как будет видно далее, в различных контекстах полезны обе точки зрения. Однако в следующей главе при рассмотрении анализа требований станет ясно, что зачастую более полезны указатели. Атрибуты — это ассоциации с более примитивными типами, которые обычно не включаются в структурную диаграмму (такие, как `String` или `Integer`). Определение "примитивных" типов — это довольно субъективное решение, поэтому формально нет никакого различия между атрибутами и ассоциациями.

Язык UML позволяет добавлять к классам дополнительную информацию с помощью записи {`tag=`*`значение`*}. Ниже приведены наиболее полезные дескрипторы.

- `description =` *`текст описания`*
- `keyword =` *`ключевое слово классификации`*; например, ботанический и т.д.
- `object_classification = domain|application|interface`
- `stereotype =` *`дополнительный стереотип`*; например, `deferred` (отсроченный), `role` (роль) и т.д.

Имена абстрактных классов выделяются курсивом. В спецификации, когда речь идет о типах, а не о классах, понятие "абстрактный", разумеется, не имеет смысла.

Язык UML поддерживает следующие аннотации к ассоциациям.

- значение по умолчанию (или начальное значение) (*`имя : Тип=выражение`*)
- префикс видимости (visibility prefix) ("+"— означает `public`, "-" —`private`, "#" — `protected`)

Примечания или соглашения об именовании можно использовать для добавления дополнительных аспектов следующих типов.

- Атрибут может иметь список допустимых значений (если он перечислимого типа (enumeration type)) и вводную часть запроса (query preface) (текст, используемый для какого-либо запроса по этому атрибуту).

- Атрибут может представлять одно из множества возможных состояний объекта.
- Типы ассоциаций определяются как множество, множество с повторяющимися элементами, упорядоченное множество или список.
- Атрибуты могут быть переменными, фиксированными, общими и уникальными. **Фиксированный** атрибут означает, что его значение не изменяется в течение всей жизни объекта. Различные экземпляры могут иметь разные значения. **Переменный** атрибут является противоположностью фиксированному атрибуту и имеет значение по умолчанию. Для **общих** атрибутов требуется, чтобы все экземпляры имели одинаковое значение. При этом его знать не обязательно. **Уникальные** атрибуты являются противоположностью общим атрибутам. Каждый экземпляр имеет свое значение. Широко известным примером является первичный ключ в таблице базы данных. Значение по умолчанию не является ни общим, ни уникальным. Обозначение для атрибута может быть одним из следующих: `{variable}`, `{fixed}`, `{common}`, `{unique}`, `{fixed,common}`, `{fixed,unique}`, `{variable,common}`, `{variable,unique}`[2].
- Может быть определен уровень безопасности.
- С помощью тегированного значения (tagged value) может быть определен владелец (ownership).
- Могут быть разрешены или запрещены неопределенные значения. Если они запрещены, то аспект (facet) `NON-NULL` устанавливается равным истине (true). Для ассоциаций это представляется с помощью минимальной единичной кратности, например `WorksFor: Dept (1, n)`.
- Может быть определено допустимое ограничение для диапазона значений, например `age> 16`.
- Символ `$` перед именем атрибута определяет атрибут класса. Его отсутствие указывает, что этот атрибут является атрибутом экземпляра. Атрибут класса является свойством всей коллекции экземпляров класса, например таким свойством, как максимальный рост людей. Атрибут экземпляра может иметь разное значение для каждого экземпляра (например, рост человека).
- Символ `x` перед именем атрибута указывает, что его значение не может наследоваться.
- Символ `/` перед именем атрибута указывает производный, т.е. унаследованный атрибут.

Операции являются спецификациями методов объекта. Язык UML поддерживает следующие аспекты операций.

- область видимости ("+" означает `public`, "-" — `private`, "#" — `protected`)
- протокол и тип возвращаемого значения (`name(argl:Type..., argN: Type): Type`)

---

[2] Эта трихотомия возникла благодаря Деннису де Чампо и его соавторам [152].

Эти аспекты могут использоваться также для определения области видимости пакетов. Как будет видно далее, весьма важными являются дополнительные аспекты, например, предусловия (pre-conditions).

6.3.1. Объектные структуры

Теперь необходимо графически проиллюстрировать, каким образом объекты связаны между собой. В главе 1 уже встречались некоторые виды обозначений. Существует четыре основных способа взаимодействия объектов. Наиболее простым из них является ассоциация, которая указывает, что тип определен на основе другого типа.

Ассоциация

На рис. 6.5 с помощью обозначений языка UML показаны ассоциации. Имена ролей (rolename) могут рассматриваться как обозначения для ограничений кратности (cardinality constraint) или как атрибуты типа. Например, `holder` можно рассматривать как атрибут `Account`, ограничивающий пределы типа `Customer` значениями 1 и 2.

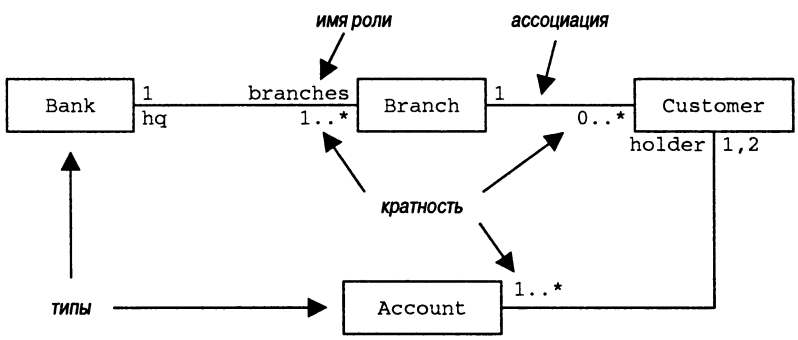


Рис. 6.5. Ассоциации в языке UML

Если ассоциация обладает интересными свойствами, она может быть представлена в виде нового типа с соответствующими атрибутами и двумя новыми ассоциациями по отношению к типам, которые она первоначально связывала. Например, ассоциация `married-to` между типами `Man` и `Woman` в системе учета сотрудников была бы простой, но для системы регистрации бракосочетания она должна быть типом, атрибутами которого являются имена вступающих в брак людей, дата бракосочетания и т.д. Язык UML имеет специальное обозначение для таких "классов ассоциаций", но оно совершенно не относится к основной теме данной книги и поэтому здесь не будет использоваться (для получения более подробных сведений см. приложение B).

Как уже обсуждалось в разделе 5.2.1, типы ассоциаций не должны вводиться просто для удаления связей "многие ко многим", как это обычно делается при проектировании реляционной базы данных. Они должны использоваться только там, где им можно дать понятные имена. В качестве контрпримера можно привести связь `PRODUCT-REGULATION`, которая уже рассматривалась в главе 5.

Преобразование ассоциаций в типы можно также использовать для введения отличий между ролями и участниками и определения типов, которые играют роли. Для иллюстрации этого на рис. 6.6 показан набор вариантов или усовершенствований одной и той же модели. Обратите внимание на то, что понятие трудоустройства может быть материализовано в типе `Job` (Работа). Преобразование этого типа обратно в ассоциацию позволяет выявить роли исполнителей. Вряд ли это стоит делать во время определения спецификации, но важно знать, что это возможно, потому что во время разработки это поможет спроектировать классы. Более детально эти вопросы будут рассматриваться в главе 9.

В [333] показано, что двунаправленные ассоциации, изображенные на рис. 6.5 и 6.6, нарушают инкапсуляцию и, таким образом, препятствуют повторному использованию[3]. Этот вид диаграммы пригоден для схематического изображения предварительных моделей и определения словаря предметной области, но при описании компонентов многократного использования предпочтительнее рассматривать ассоциации как однонаправленные указатели, которые соответствуют названиям ролей (rolenames) на рисунке. Это станет более понятным при обсуждении инвариантов.

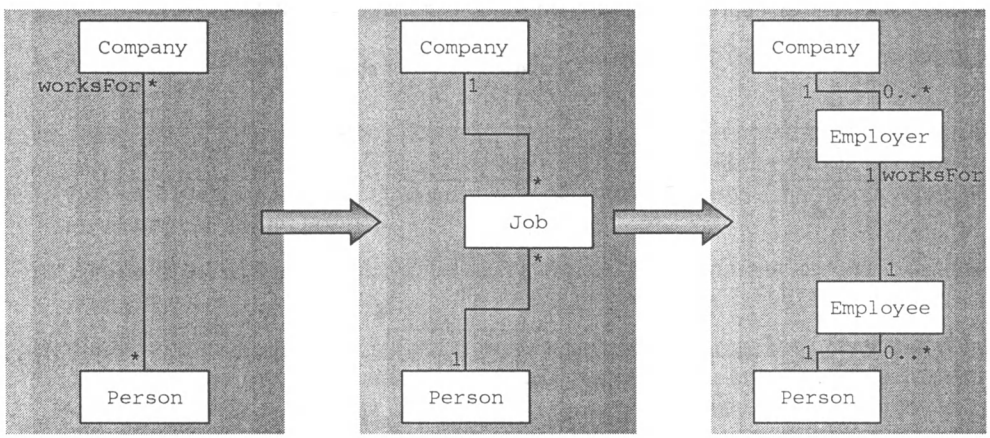


Рис. 6.6. Различие ролей и участников[4]

Понятие интерфейса объекта, содержащего атрибуты и другие аспекты, уже вводилось выше. Теперь можно рассмотреть ассоциации в качестве (обобщенных) атрибутов. В этом смысле атрибут содержит нуль или более значений некоторого типа. Ассоциация содержит значения типа, определяемого пользователем (или библиотекой). Единственным различием есть то, что атрибуты указывают на "примитивные" типы. Что является примитивным, решает аналитик. Ответ на этот вопрос — это основной критерий для представления данного типа (класса) на диаграммах классов.

Типичные ассоциации в приложении могут показывать, что служащие должны работать только в одном отделе, а отделы могут включать нуль или более служащих. Это можно описать следующим образом.

---

[3] Это будет аргументировано ниже.

[4] Этот и несколько последующих рисунков в этой главе использованы с разрешения фирмы TriReme International Ltd.

`worksIn: Dept(1,1)` является атрибутом `Employee`

`employs: (Employee,0,n)` является атрибутом `Dept`

Атрибуты разделяются на **чистые атрибуты** (pure attribute) и **ассоциации** (association). С формальной точки зрения между ними нет никакого различия, но связи чистых атрибутов не показаны на диаграммах ассоциаций, главным образом во избежание путаницы. Мощность чистого атрибута по умолчанию соответствует множеству (0,1). Если атрибут является ненулевым (non-null), это нужно указать как один из аспектов.

Ассоциации являются направленными

Это определение ассоциаций немного отличается от определений, данных в таких методах, как OMT, Шлеер-Меллора (Shlaer-Mellor), Коада (Coad), или в языке UML. В этих методах ассоциации рассматриваются как внешние по отношению к объектам, и их метамодели требуют новых обозначений. Это наиболее очевидно при использовании двунаправленных ассоциаций.

Как уже указывалось, одним из двух основных характерных свойств объектной ориентации является принцип инкапсуляции. Он подразумевает, что объекты скрывают реализацию своих структур данных и их обработку, а обращение к объектам выполняется только через их интерфейс. Объектно-ориентированная модель использует типы объектов с целью представления понятий, делит эти типы (или их реализации в виде классов) на общий интерфейс, представляющий обязанности этих типов, и на реализацию, которая полностью скрыта от всех других частей системы. Преимущества этого подхода заключаются в том, что при этом локализуется поддержка классов и облегчается их многократное использование через обращение только к интерфейсам. Модификации классов исключаются с помощью создания специализированных подклассов, которые наследуют основные обязанности и могут заменить (переопределить) базовые классы.

Двунаправленные ассоциации нарушают инкапсуляцию. Утверждение, что класс A ассоциирован с классом B тем или другим способом, является формулировкой знания, которое касается *обоих* классов. Существует три очевидных подхода к хранению этого знания.

- Если знание отделить от обоих классов, то необходимо возвратиться к системе типов объектов первого и второго классов, например к такой системе, в которой предполагается семантическое моделирование данных. Это означает, что при многократном использовании одного из двух классов, A или B, необходимо иметь знание, которое является внешним по отношению к обоим классам и гарантирует, что важная информация о связи будет перенесена в новую систему. Так как это знание не является частью классов, оно может быть потеряно, забыто или пропущено нетерпеливым разработчиком.
- В качестве альтернативы можно было бы поместить знание внутри одного из типов объектов, скажем в A. Это также неэффективно, поскольку B утратит связь с A, что исключает возможность его успешного повторного использования там, где эта связь является существенной.
- Наконец, можно сохранить знание и в A, и в B. Этот подход ведет к общеизвестным проблемам, возникающим вследствие поддержки двух копий одной и той же информации, и обычно не допускается.

Таким образом, отделение объектов от связей нарушает инкапсуляцию и усложняет возможность повторного использования. Однако автор позже продемонстрирует, как знание действительно может быть разбито между двумя типами без потери целостности с помощью инвариантов, инкапсулированных в объектах.

Еще один путь нарушения инкапсуляции — поместить комментарии об ассоциации, включающие ограничения, на диаграммах типов, а не инкапсулировать их в интерфейсы. Способ связи объектов может быть представлен на диаграммах UML рядом с ассоциацией, к которой он относится. Безусловно, никакой класс их не инкапсулирует. Чтобы убедиться, насколько это глупо и ненужно, можно рассмотреть ограничение {or}, показанное на рис. 6.7, *а*. Этот пример на самом деле взят из документации по языку UML (www.rational.com). Он показывает, что владельцем счета в банке может быть человек или организация, но не оба сразу.

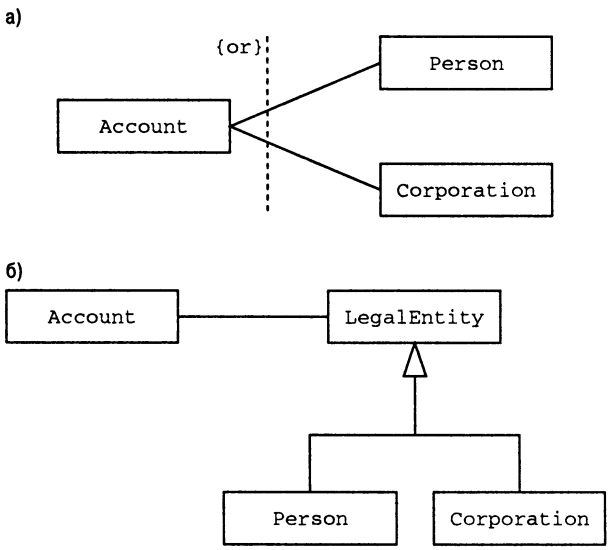


Рис. 6.7. Ограничение в языке UML, нарушающее инкапсуляцию (а), и пример использования полиморфизма для устранения необходимости ограничений (б)

Любой разработчик объектно-ориентированных систем должен отлично знать, что вместо этого неправильного проектного решения для представления дизъюнкции необходимо использовать полиморфизм наследования, как показано на рис. 6.7, *б*[5]. Здесь представлена та же информация, но инкапсуляция сохранена. Кроме того, добавлен класс многократного использования, и, по мнению автора, такое проектное решение является шаблоном анализа. В [283] часть этого решения названа шаблоном `Extract Superclass`.

[5] Можно также с тем же самым результатом специализировать класс `Account`.

Наследование

Одним из специальных видов ассоциаций в объектно-ориентированных моделях является ассоциация *обобщения* (generalization association). Ассоциация обобщения — это атрибут класса, значением которого является список классов (или типов), от которых класс наследует элементы определения. Диаграммы этих ассоциаций называют **структурами наследования**. На рис. 6.8 представлены некоторые из них. Следует отметить, что наследование может быть одиночным или множественным, как уже упоминалось в главе 1. Также необходимо отметить, что наследование типа можно отличить от наследования класса, поскольку при наследовании типа сохраняется только спецификация, а не реализация. В языке UML для индикации наследования интерфейса (вызывающего реализацию) нет специального обозначения; используется пунктирная линия со стрелкой-указателем (рис. 6.8).

Ассоциации могут быть унаследованы, что указывается с помощью косой черты (/) на линии ассоциации после имени роли и/или перед именем атрибута. Хорошие CASE-средства допускают автоматическую отмену или воспроизведение унаследованных ассоциаций.

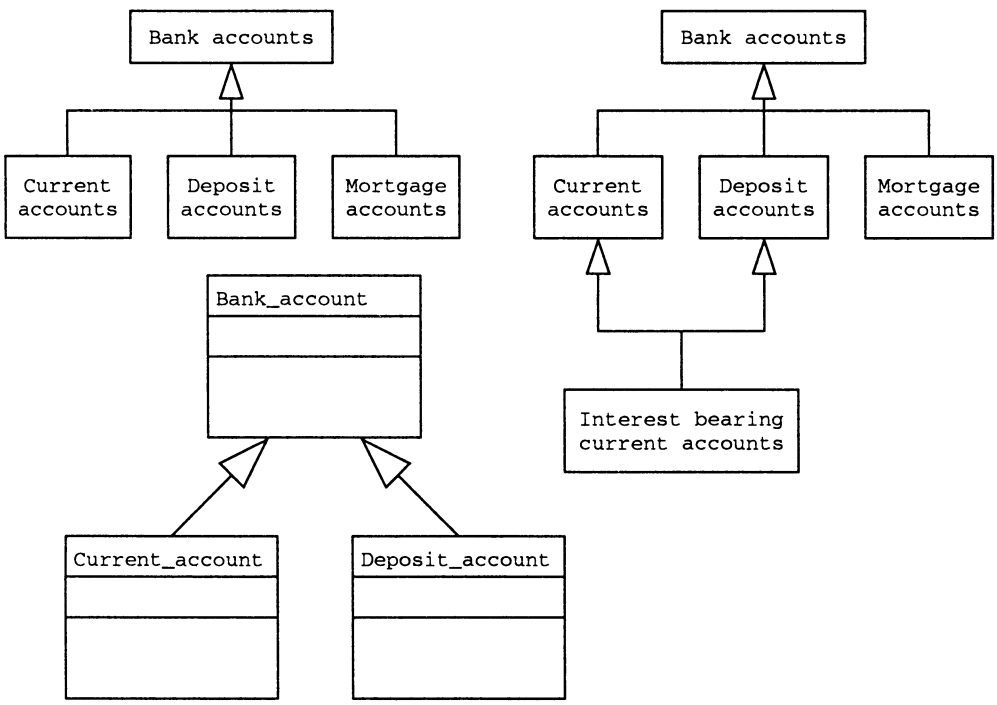


Рис. 6.8. Одиночная и множественная структуры наследования в языке UML

К сожалению, как показано в [790], невозможно привести никаких общих утверждений о способе наследования кардинальности ограничений. Можно лишь утверждать, что он зависит от интерфейса подтипа в целом. Это, разумеется, связано со способом интерпретации диаграммы как модели предметной области или как части описания требований. Таким образом, следует сделать самое слабое предположение и считать, что унаследованной кардинальностью

является нуль ко многим. Дополнительные знания о подтипе и предметной области позволяют усилить это предположение. Например, если все служащие с полной занятостью работают только в одном отделе, то служащим, занятым неполный трудовой день, можно позволить работать в нескольких отделах, но, конечно, они должны работать не менее чем в одном отделе. Об этом более подробно будет говориться при рассмотрении инвариантов.

Агрегация и композиция

Еще один специальный вид ассоциации — это агрегация или композиция. Она возникает в том случае, когда между экземплярами типов существует отношение "целое-часть". Однако необходимо использовать ее с большой осторожностью и четко осознавать, какие действия выполняются. Это будет видно при рассмотрении прецедентов (use case).

Агрегация или композиция означает, что целое состоит (конструктивно) из частей. Хорошим эвристическим тестом для проверки, является ли ассоциация композицией, служит вопрос: "Если часть перемещается, то можно ли сделать вывод, что при нормальных обстоятельствах целое перемещается вместе с ней?". Например, ассоциация "управляющий директор (УД)" между служащими и компаниями не является связью композиции, потому что управляющий директор может поехать во время отпуска в Альпы, но компания не поедет вместе с ним. С другой стороны, если ноги УД шагают по Альпам, то он тоже идет.

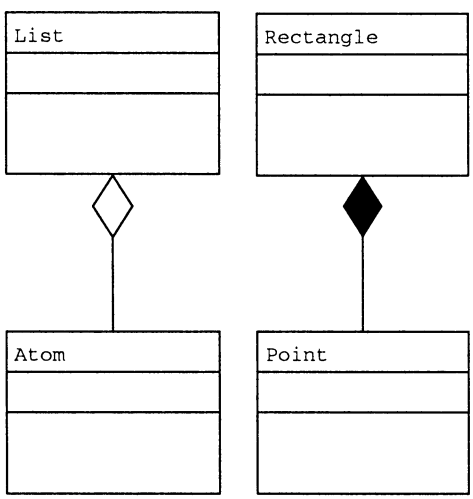


Рис. 6.9. Агрегация и композиция в языке UML

В языке UML агрегация и композиция изображаются соответственно пустым и закрашенным ромбами, как показано на рис. 6.9, и представляют понятия уровня языка программирования. В языке UML пустой ромб агрегации означает, что целое содержит *ссылку* на свою часть, а не саму часть. Это эквивалентно ссылке или конструкции указателя в языке C++. Закрашенный ромб означает, что целое отвечает за создание своих "частей". В языке C++ это реализуется следующим образом. При выделении памяти или объявлении класса-целого вслед за конструктором целого вызываются конструкторы составных частей [756]. Автору ясно, что это имеет мало общего с анализом бизнес-объектов, а также не дает определения

композиции в терминах ограничений принадлежности и времени жизни экземпляров [675]. Автор, руководствуясь здравым смыслом, продолжает использовать для составных компонентов термины композиция и агрегация. Семантика этого понятия детально исследована в [607]. Рассуждения автора этой работы можно резюмировать следующим образом.

Оделл классифицирует связи композиции по трем критериям: представляют ли они структурные связи (конфигурационное (configurational) решение), относятся ли части к тому же типу, что и целое (гомеомерное (homeomeric) решение), и могут ли части быть отделены от целого (инвариантное (invariant) решение). При этом он выделяет 8 типов решений. Затем он рассматривает шесть из них и дает свои различные интерпретации композиции.

1. "Компонент-целое" (component-integral) — конфигурационная, негомеомерная и неинвариантная композиция.
2. Вещественная (material) — конфигурационная, негомеомерная и инвариантная композиция.
3. Блочная (portion) — конфигурационная, гомеомерная и неинвариантная композиция.
4. Пространственная (place-area) — конфигурационная, гомеомерная и инвариантная композиция.
5. Групповая (member bunch) — неконфигурационная, негомеомерная и неинвариантная композиция.
6. Партнерская (member-partnership) — неконфигурационная, негомеоморфная и инвариантная композиция.

В данном разделе используется только конфигурационная инвариантная композиция. Все другие типы композиции описываются либо ассоциациями, либо атрибутами (которые являются просто специальным случаем ассоциаций). Дополнительные детали не привносят новой информации для программиста. Автор предпочитает рассматривать агрегацию как форму уточнения: составляющие — это то, что обнаруживается при более близком рассмотрении.

Как и наследование, композиция является направленной. Часть не обязана знать, какому целому она принадлежит, поскольку это усложнило бы возможность повторного использования. Чтобы подчеркнуть это, автор снабдил пример составной (composite) яхты на рис. 6.10 стрелками-указателями. Там также показано, что каждая композиционная связь может иметь ограничение на мощность (cardinality constraint). Указание подобного ограничения со стороны целого не только нарушает инкапсуляцию, но и требует разграничения композиции на уровне типа и экземпляра (как это было сделано в предыдущих изданиях этой книги). В настоящее время автор полагает, что лучше всего использовать понятие "тип мощности" (power type), введенное в [524]. Тип мощности — это такой тип, экземпляры которого являются подтипами другого типа. Типы могут иметь несколько типов мощности, соответствующих отдельным подтипам. Например, если служащие разделяются на мужчин и женщин, то можно создать (немного избыточный в этом случае) тип мощности с именем `GenderType`; но если классифицировать их в соответствии с типом работы, то может использоваться другой (и возможно, более полезный) тип мощности. Вернемся к агрегации. Если рассмотреть велосипед, состоящий из рамы и нескольких (двух?) колес, то конкретное колесо может принадлежать только одному велосипеду. Однако тип колеса может быть пригоден для нескольких типов велосипедов. Если не вводить тип мощности, соответствующий описанию колес `WheelType`, то это приведет к необходимости отличать агрегацию на уровне типа и на уровне

экземпляра. Этот феномен называется рефлексией (reflection). Обычно этот термин используется для описания возможности расширения модели "во время выполнения". Идея заключается в том, что аналитик определяет модель, в которой имеется тип типов, который может быть использован во время выполнения для добавления новых типов, но его связи, тем не менее, определяются на основном языке спецификации.

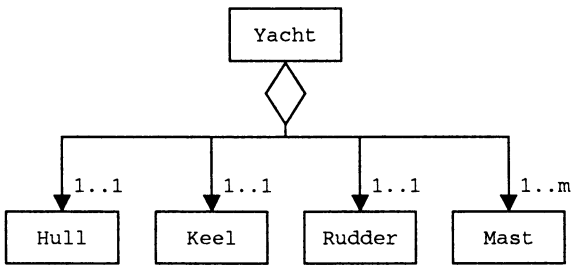


Рис. 6.10. Композиционная структура яхты

Подобно другим видам ассоциации, композиция может быть унаследована, и чтобы показать это, используется то же соглашение, что и для обычных ассоциаций. Это знак "/" на связи. Подтип (subtype) наследует компоненты всех своих супертипов (supertype). Рассмотрим тип A с подтипами B и C. В данном случае A имеет компоненты A1 и A2. Класс B имеет компоненты B1 и B2. Следовательно, иерархия композиции для класса B включает два производных компонента, как это показано на рис. 6.11. Полученные зависимости для большей ясности изображены серым цветом. Автор не показал производные зависимости для C. Теперь хорошо видно, каким образом включение всех производных связей может сделать диаграммы нечитабельными. Поэтому CASE-средства должны предоставлять возможность их включения и отключения.

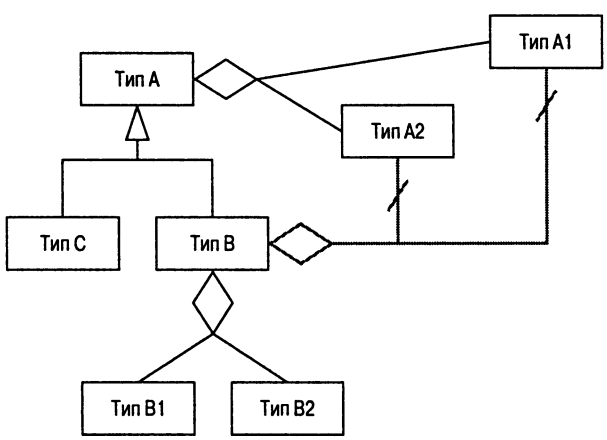


Рис. 6.11. Производные зависимости композиции

Одни специалисты рассматривают понятие композиции как неопределяемое, а другие — как определяемое только в терминах некоторого конкретного набора операций, определенных для целого. Автор рекомендует очень осторожно использовать композицию при моделировании бизнес-объектов. Но как будет видно из следующего раздела и главы 7, ее использование является абсолютно необходимым при моделировании объектов-действий.

Зависимости

Язык UML также допускает использовать зависимости произвольного вида между типами и классами. Они изображаются пунктирной линией со стрелкой на конце и меткой. Метка — это стереотип. Наиболее привычные и полезные зависимости касаются интерфейсов и пакетов. Они будут рассматриваться далее в соответствующих разделах.

Использование

Связи **использования** (usage) означают, что клиент не только знает о существовании некоторой возможности сервера, но и в действительности пошлет сообщение, чтобы применить в какой-то момент эту возможность. Простые ассоциации могут никогда не применяться. Различие между ассоциацией и зависимостью использования (usage dependency) аналогично различию между знанием адреса редактора газеты *Financial Times* и работой корреспондентом на Уолл Стрит. Это понятие не относится к реализации, как утверждают многие из критиков автора, но составляет основную часть семантики модели. Высказывание о том, что два класса связаны, не подразумевает наличия структурной связи между ними. Например, в базе данных может быть много связей, необходимых для поддержки в ней *нерегламентированных* запросов (*ad hoc* query), которые, возможно, никогда не будут сделаны ни одним из пользователей. С другой стороны, связь использования констатирует некоторое взаимодействие или сотрудничество. Существование связей использования устраняет необходимость в диаграммах взаимодействия [803]. Один класс "использует" (uses) другой, если он является клиентом другого класса, выступающего в качестве сервера. Любые введенные ассоциации впоследствии могут быть заменены более конкретной связью использования или (реже) композицией. Этот вид связи также чрезвычайно важен в разработке требований и моделировании бизнес-процессов, что будет видно из главы 7. Это понятие отсутствует как в языке OML, так и в языке UML, хотя в UML для представления этой идеи можно использовать зависимость с меткой "использует". В [375] приводятся доводы в пользу включения связи "использует" в язык OML.

На рис. 6.12 показаны легко запоминаемый значок для представления стереотипа "использует" и более привычное обозначение.

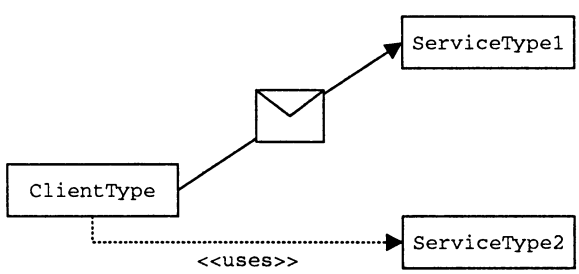


Рис. 6.12. Ассоциации использования

6.3.2 Применение прецедентов для нахождения типов

До сих пор рассказывалось, в основном, об изображении объектов, но почти не говорилось о том, каким образом их выявлять. Наиболее популярная методика выявления объектов состоит в описании множества прецедентов, определяющих, каким образом система взаимодействует с внешней средой и пользователями.

В методе Catalysis представлена концепция **действий**, которая обобщает понятия прецедентов и операций. **Действие** (action) — это любое целенаправленное сотрудничество, деятельность, задача, операция или работа, связывающая агентов в бизнесе, системе или проекте. При спецификации систем самыми интересными действиями являются прецеденты. **Прецедент** (use case) — это целенаправленное сотрудничество между системой и действующим субъектом (исполнителем). **Исполнитель** (actor) — это пользователь, выполняющий какую-то роль. Таким "пользователем" может быть устройство или компонент, а также человек. **Агент** (agent) — это некто, заимствующий роль, в отличие от пользователя, который выполняет ее по определению. Агенты и исполнители рассматриваются в качестве объектов, поскольку они обладают состоянием, поведением и идентичностью. Если известен инициатор сотрудничества, то зависимость использования можно рассматривать как шаблон действия. Рассмотрим простое действие: покупку какого-то товара (рис. 6.13). Эллипс представляет взаимодействие между двумя экземплярами агентов, что отражено в неформальном постусловии, которое описано в примечании. Это постусловие относится только к **параметрам** объектов. Здесь товар `Thing` идентифицирован как тип, который должен быть включен в словарь. Кроме того, в постусловии выделены некоторые существительные. Это очень полезная методика, позволяющая из описания прецедентов логически вывести модель типа. Существительные, претендующие на роль типов (candidate types), показаны жирным шрифтом, атрибуты подчеркнуты, а потенциальные ассоциации выделены курсивом. Этот метод так называемого текстового анализа рассматривается далее в разделе 6.5. Он описан в [2].

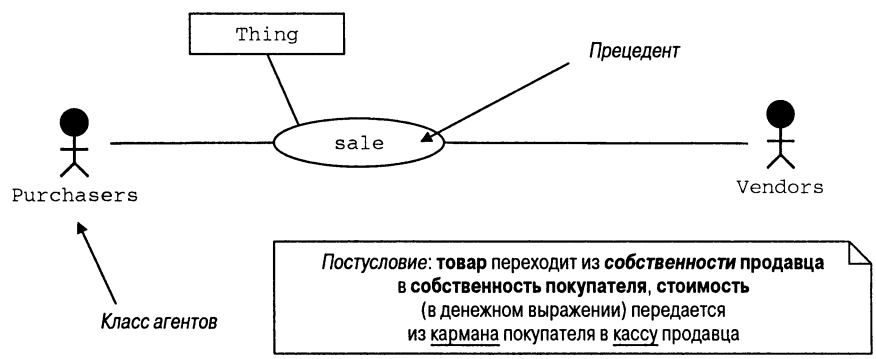


Рис. 6.13. Прецеденты и типы, которые можно выделить из их описания

Из этого примера видно, что действие всегда приводит к изменению состояния, которое может быть выражено постусловием. В методе Catalysis [204] это изменение проиллюстрировано с использованием **моментальных снимков экземпляра** (instance snapshot). Экземпляры типов-кандидатов и их ассоциации изображаются в определенный момент времени перед наступлением действия. Затем на основе результатов прецедента модифицируются

ассоциации на диаграмме (рис. 6.14). Ассоциации, присутствующие на более поздних диаграммах, показаны в этой книге толстыми серыми линиями. Следует обратить внимание также на то, что значения атрибутов `pocket` (карман) и `till` (касса) вычеркнуты и заменены новыми.

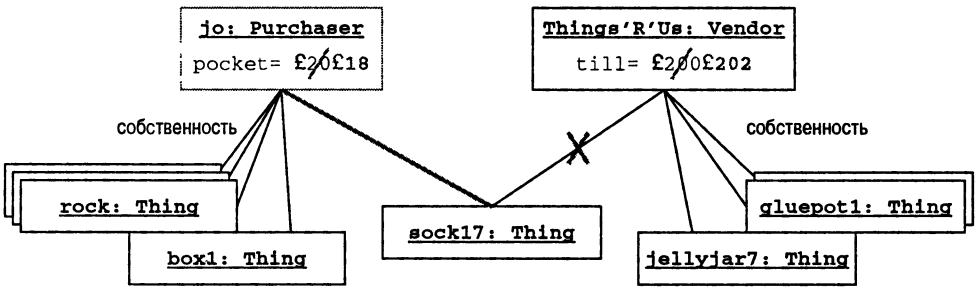


Рис. 6.14. Моментальный снимок экземпляра

Любой моментальный снимок должен соответствовать диаграмме типа, ассоциированной с иллюстрируемым прецедентом. Моментальные снимки полезны в качестве поддержки абстрактных рассуждений на этапе анализа и проектирования. Они представляют экземпляры прецедентов. Хотя простые моментальные снимки могут быть включены в официальную документацию, автор не поддерживает такие решения. Модель типа может быть со временем модифицирована. В особенности следует обратить внимание на то, что ни постусловие, ни моментальные снимки ничего не говорят о последовательности событий; они только описывают результат. Это важно при переходе к проектированию компонентов многократного использования, поскольку два компонента (или два бизнес-модуля) могут реализовывать прецедент совершенно по-разному. Интерфейс компонента многократного использования должен определять только результат и терминологию, но не последовательность операций выполнения, которая приводит к этому результату.

Для каждого действия и, тем более, прецедента указываются название, цель, список участников и список параметров — объектов (вовлеченных активно или пассивно), которые различны для каждой реализации (occurrence) прецедента. Цель включает результаты выполнения этого прецендента и/или условия его реализации. Можно задавать предусловие: утверждение, определяющее, при каких условиях имеет смысл этот прецедент. Таким образом, при документировании действия рекомендуется следующая форма записи.

```
тип имя (параметры)
предусловие
постусловие
```

Например, в соответствии с рис. 6.13 можно записать следующее.

```
прецедент buy_thing (Purchaser, Vendor, Thing)
   предусловие: у продавца есть товар, который может понадо-
биться покупателю
   постусловие: у продавца количество товаров уменьшилось
   и  количество товаров у покупателя увеличилось
```

```
и  vendor.till += thing.price
и  purchaser.pocket -= thing.price
```

Используя язык OCL (Object Constraint Language — объектный язык ограничений), это можно формализовать следующим образом.

```
use case buy_thing (Purchaser, Vendor, Thing)
   pre:   vendor.possessions -> includes thing
   and    purchaser.pocket >= thing.price
   post:  vendor.possessions = vendor.possessions - thing
   and   purchaser.possessions = purchaser.possessions + thing
   and   vendor.till = vendor.till@pre + thing.price
   and   purchaser.pocket = purchaser.pocket@pre -thing.price
```

Приведенное выше выражение `@pre` означает предыдущее значение атрибута, а `->includes` указывает на принадлежность множеству. Знак "+" обозначает `->union`. Еще одно используемое здесь соглашение унаследовано от метода Fusion [179]: если все параметры имеют различные типы, то их имена совпадают с именами типов, но начинаются со строчной буквы. Это же соглашение используется для непомеченных ассоциаций. Использование прописных букв означает тип. Также следует обратить внимание на отсутствие атрибута `price` у объекта `Thing`. Единственная задача атрибутов состоит в том, чтобы обеспечить терминологию для спецификации действий.

Спецификации языка OCL нравятся не всем, и их можно не использовать. Но возможность такой формализации играет важную роль в системах с высокой степенью целостности (high-integrity system) или с особыми требованиями к безопасности (safety-critical system). Автор этой книги использует язык OCL в небольшом количестве.

Статическая модель типов содержит терминологию для описания прецедентов. Модель прецедентов определяет поведение, используя термины статической модели. Моментальные снимки иллюстрируют реализацию прецедентов и помогают в процессе уточнения модели типов.

Прецеденты (и другие действия) являются объектами, поэтому они могут быть ассоциированы, классифицированы и объединены точно так же, как и другие объекты. Из рассмотренного примера видно, что могут существовать различные виды продаж, которые конкретизируют это понятие. Следует обратить внимание на то, что все постусловия усилены в подчиненных действиях (sub-action). Предусловия, если таковые имеются, должны быть ослаблены. Ниже будет видно, что это является общим правилом для наследования пред- и постусловий. Прецеденты можно обобщить и уточнить, используя агрегацию (рис. 6.15) и наследование (рис. 6.16). На рис. 6.16 представлен пример специализации продажи. Следует обратить внимание на то, что специализированный прецедент достигает всех целей своего(их) порождающего(их) элемента(ов).

На рис. 6.16 представлена типичная ошибка. Совершенно неправильно использовать одну и ту же операцию наследования для описания несопоставимых понятий: продажа бензина (petrol sale) и продажа в кредит (credit sale). Необходимо отдельно показать две ортогональные иерархии; каждая операция наследования соответствует своим **дискриминаторам** (discriminator), таким как "тип топлива (fuel type)" или "метод оплаты (payment method)". При построении диаграммы специализации всегда необходимо задавать себе вопрос: "Что является дискриминатором?".

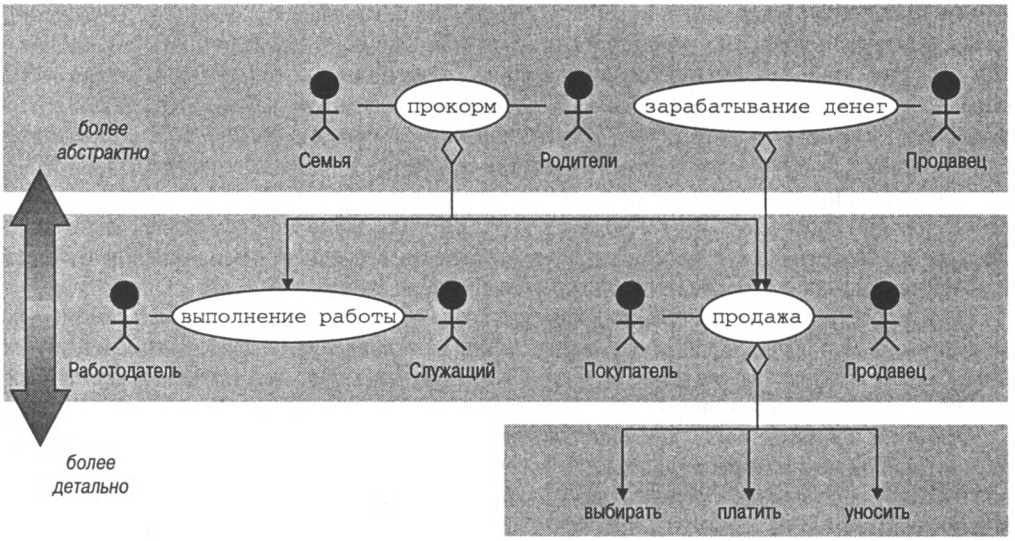


Рис. 6.15. Композиция и декомпозиция прецедентов

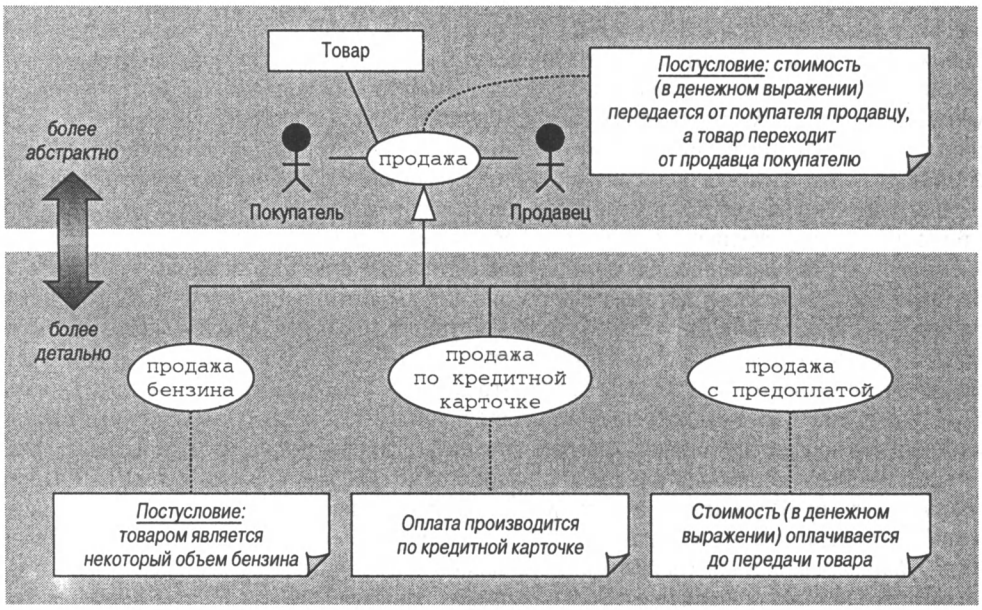


Рис. 6.16. Обобщение и специализация прецедентов

Упорядочение элементов диаграммы агрегации слева направо, как на рис. 6.15, не подразумевает какой-либо временной последовательности их выполнения. Прецеденты могут реализовываться в любой последовательности, могут повторяться или быть параллельными.

Тем не менее можно добавлять подобную информацию, создавая ассоциации между действиями, как это можно будет увидеть в главе 7, где также обсуждается обработка исключительных ситуаций с применением зависимости `<<uses>>` (использует). В языке UML определены две конкретные зависимости, которые обозначены как `<<include>>` (первоначально `<<uses>>`) и `<<extend>>` (первоначально `<<extends>>`). Однако поскольку они определены неудачно и противоречиво и `<<extend>>` нарушает инкапсуляцию [333], автор в этой книге их не будет использовать. Основываясь на опыте, автор полагает, что их использование вносит путаницу. Кроме того, стандартная объектная семантика, которая установлена для статических моделей, дает возможность все описать без введения дополнительной терминологии. Более подробно об этом будет сказано в главе 7.

Цели прецедентов обеспечивают основу для создания средств тестирования в самом начале анализа, поскольку они непосредственно связаны с назначением и функциями системы (об этом больше будет сказано в главе 8). Этому также способствует экстремальное программирование (eXtreme Programming) [63].

Обратите внимание на то, как исполнители в модели прецедентов соответствуют типам в рассмотренной ранее модели типов. Если рассматривать модель типов с точки зрения терминологии для определения прецедентов, то можно заметить связь между двумя различными видами диаграмм UML. Эта связь впервые была показана в методе Catalysis.

Точно так же, как моментальные снимки помогают визуализации модели типов, сценарии и диаграммы последовательности помогают визуализации прецедентов. Сценарии — это описания бизнес-процессов. Обычно они излагаются в виде текстового описания и представляют типичные последовательности действий при реализации прецедента, но могут включать вариации основной последовательности. Они могут быть проиллюстрированы диаграммами последовательности или сотрудничества UML, например, такого вида, как на рис. 6.18, или, что лучше всего, диаграммой последовательности в стиле метода Catalysis, как на рис. 6.17. Отличительная особенность последнего — это экземпляры действий (action-instance), каждый из которых может касаться более двух объектов и не обязательно является направленным.

Диаграммы последовательностей UML описывают только сообщения ООП, каждое из которых имеет получателя и отправителя. В обоих случаях вертикальные линии представляют экземпляры типов, указанных над ними, а горизонтальные линии представляют действия, затрагивающие экземпляры, которые они соединяют. Оба измерения могут быть расширены для описания действий более мелкого и более крупного масштаба; каждый прецедент может быть расширен до более подробных последовательностей, а каждый объект — до более детализированных объектов. Из рисунка видно, что детальное изображение прецедента (покупки барабана) можно дать крупным планом, чтобы показать в деталях последовательность бизнес-процесса продавца. Можно также дать крупный план объекта `Vendor` (продавец), чтобы показать детали организации его работы. Необходимо выбрать уровень и масштаб работы. *Основной* (essential) уровень соответствует следующему описанию: "Моя зарплата записана на мой счет вчера, так что я пошел, чтобы сегодня получить некоторое количество наличных денег". *Детализированное* (detail) описание можно представить так: "Чтобы получить деньги со счета, Вы вставляете вашу карточку в банкомат, вводите PIN-код ...". Вот пример *укрупненного описания*: "Хороший способ делать деньги — это управлять банком. Обеспечить счета более высокими процентами в обмен на меньшую доступность. Обеспечить возможность снятия наличных денег с наиболее доступных счетов в любое время дня или ночи." Далее в главе 8 будут рассмотрены основные прецеденты.

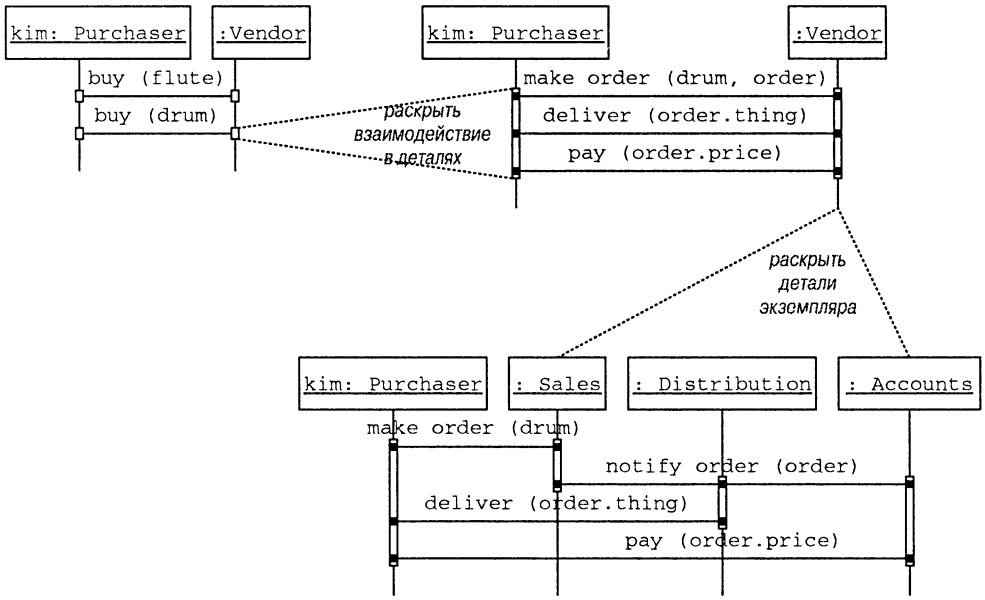


Рис. 6.17. Диаграммы последовательности и их уточнение

Диаграммы последовательности для сценариев помогают идентифицировать различных участников действия и их взаимодействия, а также показать последовательность событий. Это помогает формированию концептуального представления событий во времени и пониманию зависимостей между экземплярами. Диаграммы последовательности помогают разработчику распределить экземпляры среди распределенных системных компонентов и часто полезны во время тестирования. Они также помогают исследованию взаимовлияния различных прецедентов.

Следует обратить внимание на то, что в приведенных выше примерах изменение масштаба изображения основано на агрегации. Наследование также может вести к уточнению.

Уточнение — это один из основных принципов метода Catalysis, который упрощает трассировку, привязывая прецеденты уровня спецификации и типы к интерфейсам реализации. Как правило, различные уровни уточнения содержатся в различных пакетах реализации (implementation package).

Диаграммы последовательностей обеспечивают визуализацию распределения обязанностей: задачи можно просто сдвинуть в сторону. Они эквивалентны диаграммам сотрудничества, которые подчеркивают статические, а не временные отношения. Диаграммы сотрудничества помогают увидеть ассоциации и зависимости между объектами (которые, разумеется, желательно минимизировать). Диаграммы последовательностей способны иллюстрировать распределение обязанностей, но не могут адекватно отразить процесс управления. Диаграммы сотрудничества лучше подходят для отображения ассоциаций и процесса управления. Это позволяет повысить автономность объектов. На рис. 6.18 показана диаграмма последовательности и связанная с ней форма сотрудничества. Стрелки и линии, соединяющие экземпляры в форме сотрудничества, представляют ассоциации. Сообщения обозначены неприкрепленными стрелками, и их нумерация указывает последовательность их

вызова. Детализированные диаграммы сотрудничества все еще далеки от деталей самой реализации. Из рис. 6.18 видно, что ассоциация между менеджером и рабочей корзинкой `WorkTray` недопустима. Менеджеру лучше обращаться к этим данным через секретаря. Изменение способа хранения служебных приложений (job-applications) привело бы к изменению действий и секретаря `Secretary`, и менеджера `Manager`, если они оба обращаются к объекту `WorkTray`. Мораль заключается в том, что можно сократить зависимости, назначая обязанности должным образом. Диаграммы сотрудничества помогают понять, каким образом назначать обязанности. На рис. 6.18 также показано, что диаграммы сотрудничества могут быть использованы в качестве моментальных снимков.

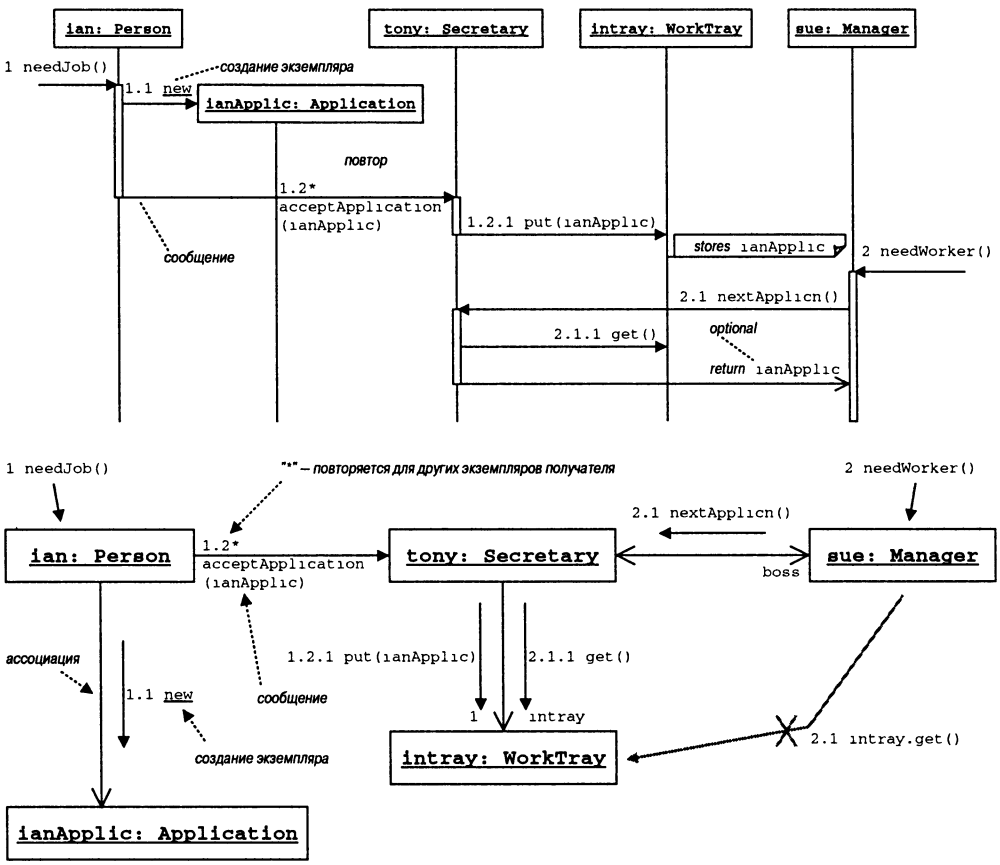


Рис. 6.18. Диаграммы последовательности и сотрудничества

6.3.3. Инварианты и наборы правил

При выполнении объектно-ориентированного анализа и моделирования можно воспользоваться результатами, полученными в области искусственного интеллекта, семантических сетей и семантического моделирования данных. Спецификации, которые отражают инкапсуляцию

атрибутов и операций, очень хороши, но не обязательно четко описывают аспекты, которые имели в виду аналитики или пользователи. Для многократного использования спецификация объекта должна содержать знания объекта (атрибуты), его действия (операции), а также мотивы этих действий и связь с интерфейсами других объектов. Автор считает, что эта семантика частично содержится в структурах ассоциаций, классификации, композиции и использования, а также в утверждениях, инвариантах и наборах правил, которые описывают поведение объекта.

Известно, что введение семантики ограничивает многократное использование, но делает его более безопасным. Например, это относится к наследованию, а также другим отношениям, конкретизирующим *значение* объекта. В системной спецификации оба аспекта одинаково важны, поэтому следует хорошо понимать все преимущества и недостатки каждого из подходов, а также учитывать цели аналитиков и их клиентов.

В процессе проектирования приходится решать, принадлежат ли правила отдельным операциям или объекту в целом. Операции могут быть выражены на языке продукционных правил (rule-based language). Однако различие состоит не в форме выражения, а в содержании правил. Правила, связывающие несколько операций, не определяют зависимости между атрибутами и относятся к объекту в целом. Наоборот, правила, которые касаются инкапсуляции состояния объекта, находятся в пределах одной из его операций. Наиболее важный вид правил "объекта в целом" — это правила управления, которые описывают поведение объекта, его участие в структурах, к которым он принадлежит. Это правила управления значениями по умолчанию, множественного наследования, исключительных ситуаций и ассоциаций с другими объектами.

Введение инкапсулированного набора правил является новым аспектом метода SOMA. Это расширяет объектные модели, добавляя множество наборов правил к каждому объекту. Таким образом, если обычный объект состоит из идентификатора (identifier), атрибутов (attributes) и операций (operations), то объект SOMA состоит из идентификатора, атрибутов, операций и наборов правил (ruleset). **Наборы правил** составлены из неупорядоченного набора утверждений и правил или в форме "если..., то...". Это расширение имеет множество интересных последствий. Наиболее примечательным из них является следующее. Объекты, являющиеся локальными сущностями, могут формировать правила для глобального системного поведения (в некоторой степени, подобно ДНК, инкапсулировать морфему). Дальнейшее следствие заключается в том, что объекты с набором правил могут рассматриваться в качестве интеллектуальных агентов для экспертных систем.

Принято общее соглашение о том, что определять только атрибуты и операции (сигнатуру) объекта совершенно недостаточно. Чтобы определить объект полностью, необходимо сказать о том, каким образом он взаимодействует с другими объектами и каким правилам и ограничениям подчиняется для обеспечения целостности. Некоторые языки, такие как Eiffel, и некоторые методы, такие как BON, Catalysis или Syntropy, достигают частичного решения, используя формальные утверждения. Утверждения в таких системах делятся на два вида: утверждения об операциях и утверждения о целом объекте. Первые — это типичные предусловия и постусловия, в то время как последние называют инвариантами класса. Методы SOMA и Catalysis добавляют условия инвариантности к операционным утверждениям, а метод SOMA обобщает инварианты класса в наборы правил, которые могут быть совместно использованы для логического вывода новой информации. Также имеются утверждения, представляющие ограничения атрибутов. Далее приводятся определения.

Утверждения атрибутов делятся на следующие виды.

- **Ограничения диапазона** описывают пределы изменения допустимых значений.
- **Ограничения перечня** перечисляют допустимые значения.
- **Ограничения типа** определяют класс, которому должны принадлежать значения. Ограничения типа всегда представляют и обобщают первые два случая.

Операционные утверждения делятся на следующие.

- **Предусловие** — это отдельное логическое утверждение, которое должно быть истинным до выполнения соответствующей операции.
- **Постусловие** — это отдельное логическое утверждение, которое должно быть истинным после того, как соответствующая операция закончила выполнение.
- **Условие инвариантности** — это отдельное логическое утверждение, которое должно быть постоянным при выполнении операции. Это имеет значение только для параллельных систем обработки (включая модели бизнес-процессов). Условия инвариантности были сначала представлены как часть метода SOMA [312]. Метод Catalysis [204] различает два вида условий инвариантности: гарантия и выражение уверенности.
- **Выражение уверенности** устанавливает предварительное условие, которое должно оставаться истинным в течение выполнения операции, к которой это относится. Если оно нарушено, спецификация не устанавливает результата. Сервер не отвечает за поддержку этого условия.
- **Гарантия** — это утверждение, которое сервер должен поддерживать как истину в течение выполнения операции.

Аспекты операции могут включать более одного утверждения любого из этих типов. Утверждения могут быть представлены в соответствии с диаграммами состояний, как это будет видно далее.

Объектные утверждения и наборы правил делятся на следующие виды.

- **Инвариант класса** — это одно логическое утверждение о любом подмножестве возможностей объекта, которое должно быть истинным всегда (в языке Eiffel, имеющем прямую поддержку инвариантов, это применяется только в те моменты, когда метод не выполняется). Ограничения кратности для атрибутов являются инвариантами. Инварианты можно также назвать **правилами**.
- **Набор правил** — это неупорядоченный набор инвариантов класса (или **правил**) и утверждений об атрибутах с определенным режимом логического вывода, который обеспечивает объединение правил. **Внешние наборы правил** выражают информацию второго порядка, такую как стратегия управления. **Внутренние наборы правил** — это множества инвариантов (первого порядка). Они могут быть написаны на естественном языке, на языке OCL или на исполняемом языке правил.
- **Эффект** — это постусловие, объединенное со всеми другими постусловиями операций типа. Эффект обычно имеет такую форму: (любое изменение $f(x, x @ \text{pre}) \Rightarrow$ условие). Эффекты могут быть выражены в виде правил. Они полезны для проектирования супертипа и наложения ограничений на операции.

Обычно, помимо сказанного выше, предполагается, что логика, лежащая в основе вышеупомянутых определений, является стандартным исчислением предикатов первого порядка FOPC (First Order Predicate Calculus). Автор не делает такого предположения, хотя FOPC — это обычная используемая по умолчанию логика.[6] Другие логики, которые могут использоваться, включают временную (temporal), нечеткую (fuzzy), этическую (deontic), понятийную (epistemic) и немонотонную (non-monotonic) логики. Каждый набор правил класса локально определяет собственную логику, хотя в одном и том же классе редко смешивают различные логики.

Различие между типом и классом уже рассматривалось: тип не имеет реализации. Теперь можно определить различие между типами и интерфейсами. **Интерфейс** — это список сообщений, которые можно передать объекту, со своими параметрами и типами возвращаемых значений. В зависимости от интерпретации, он может включать операции получения и присваивания значений атрибутам. Это понятие иногда рассматривается как **сигнатура** типа. С другой стороны, тип — это полная спецификация, включающая все утверждения, которые могут быть сделаны о типе и его экземплярах, и все наборы правил.

Инвариант или ограничение — это отдельное правило, которому всегда подчиняется объект. Оно выражается с использованием терминологии, обеспечиваемой моделью типов. Примерные инварианты для системы управления авиалинией могли бы включать следующее: "Каждый пилот должен быть капитаном или вторым пилотом одного полета в день", "Капитаном и вторым пилотом не может быть один и тот же человек" или "Каждый полет должен осуществляться капитаном, который имеет квалификацию для управления этим типом самолета". Ясно, что инвариант типа может иметь отношение к общим интерфейсам других типов. Инварианты могут быть выражены неформально, как в приведенном выше примере, или с использованием формализованного языка OCL или других формальных логических систем.

Одним из возможных источников появления инвариантов являются циклы на диаграммах типов. Фактически, как это будет видно далее, любая двунаправленная ассоциация обычно требует пары инвариантов (в точности), поскольку пара имен ролей (rolename) определяет цикл. На рис. 6.19 все циклы требуют определения одного или более инвариантов.

Например, на верхней диаграмме указано, что пилот работает на авиалинии, обеспечивающей полет. Читателю предлагается самостоятельно записать инварианты для другого цикла на рис. 6.19.

Наборы правил обобщают инварианты класса и позволяют объектам выполнять следующие функции.

- Логически выводить значения атрибутов, которые не хранятся явно.
- Представлять триггеры базы данных.
- Представлять операции непроцедурным способом.
- Представлять режимы управления.

Правила определяют информацию второго порядка, такую как зависимости между атрибутами (например, зависимость между возрастом служащей и ее правом на отпуск). Глобальные пред- и постусловия, которые применяют ко всем операциям, могут быть определены как

---

[6] В Catalysis по умолчанию используется логика частично вычислимых функций (Logic of Partial Functions) [160], которая имеет четкую трактовку неопределенных значений.

правила. Типичное бизнес-правило могло бы предполагать "изменение продолжительности отпуска до шести недель, если стаж службы превышает пять лет" в качестве одного из правил для типа `Employee`. Наборы правил позволяют решать проблемы анализа в тех случаях, когда в качестве целевой среды предусмотрена активная база данных.

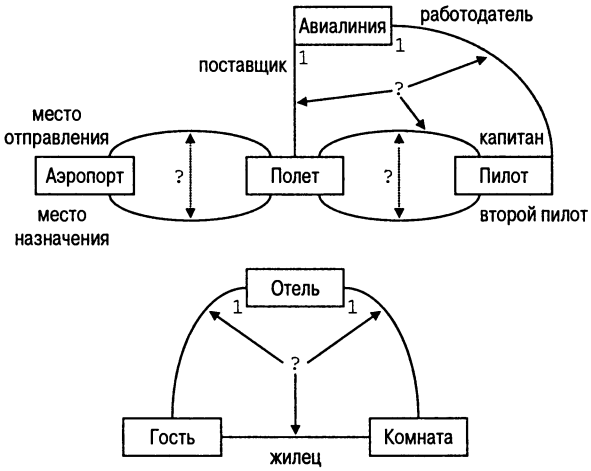


Рис. 6.19. Циклические ассоциации предполагают наличие инвариантов

Правила и инварианты используются для уточнения семантики объекта. Это помогает при описании информации, которая обычно постоянно хранится в архиве, например информации о деловой активности предприятия. Это также обеспечивает функциональную совместимость на довольно низком уровне. Например, имеется объект, который вычисляет кубические корни. Клиенту этого объекта недостаточно знать только его операции; необходимо также знать, что возвращаемое значение является кубическим, а не квадратным корнем. В этом простом случае решение очевидно, поскольку кубический корень можно уникально охарактеризовать с помощью одного простого правила: ответ, дважды умноженный на себя, равен переданному параметру. Если это правило является частью интерфейса, тогда все другие системы и системные компоненты могут понять функции объекта только из его интерфейса. Таким образом устраняются некоторые сложности технологии хранения информации за счет ее перемещения в объектную модель.

Правила, описанные в разделе правил, могут быть классифицированы на несколько, не обязательно взаимоисключающих типов.

- Бизнес-правила
- Правила обработки исключительных ситуаций
- Триггеры
- Правила управления

Правила управления

Последний тип наборов правил скрыт от внешнего доступа. Объектно-ориентированные методы должны обеспечивать множественное наследование классов. Поэтому необходимо определить способ управления конфликтными ситуациями, когда один и тот же атрибут или одна и та же операция по-разному наследуются от двух порождающих объектов. Конечно, при одиночном наследовании данная проблема не возникает. Для разрешения неоднозначности множественного наследования можно использовать наборы правил. Тогда можно также определить приоритетные правила умолчания и управления демонами (demon). (Демон — это метод, который пробуждается, когда необходимо, т.е. при изменении некоторой величины, при ее добавлении или удалении.) Правила могут устанавливать, каким образом разрешить конфликт, возникающий при наследовании атрибутом двух различных значений, или определять, когда должно быть применено значение по умолчанию (до или после наследования либо до или после срабатывания демона). Они могут также определять соответствующие приоритеты наследования и демонов. Как и в случае с атрибутами и операциями, интерфейс объекта отображает только имя набора правил. В случае обратной цепочки набора правил он может включать имя неизвестной величины, например `If Route: needed SEEK Route:`.

Правила управления инкапсулируются в пределах объектов, а не объявляются глобально. Они также могут наследоваться и отменяться, хотя в некоторых случаях правила для этого могут быть немного сложнее [790]. Преимуществом этого является возможность локальных изменений в стратегии управления. Кроме того, аналитик может исследовать воздействие структуры управления на каждый объект (возможно, используя броузер), поэтому нет смысла комментировать замысловатые диаграммы, чтобы описать локальные эффекты управления. Подлинно глобальные правила могут содержаться в объекте верхнего уровня, скажем `Object`, и наследоваться всеми объектами, не отвергающими эти правила. В качестве альтернативы можно установить глобальные правила в специальном объекте `Policy blackboard` (Доска объявления политики). Соответствующие классы регистрируют свое участие в классе `Policy`, который передает изменения правил и состояний по мере необходимости зарегистрированным классам. В данном случае, конечно, используется шаблон Publish and subscribe (см. главу 7). Подобно использованию диаграмм переходов для описания процедурной семантики операций, в описании сложных наборов правил могут быть полезными деревья решений.

Правила управления касаются операций и атрибутов объекта, которому они принадлежат. Самих себя они не касаются. Таким образом, они не могут помочь в определении того, как разрешить конфликт множественного наследования между наборами правил или другую проблему стратегии управления, связанную с наборами правил. Это потребовало бы инкапсуляции множества метаправил, для которых в свою очередь потребовался бы метаязык. Это стремительно ведет к бесконечному регрессу. Поэтому множественное наследование правил не дает возможности разрешения конфликтов. Для доступа к одноименным наборам правил используется операция `.` (точка). Таким образом, если объект унаследовал наборы правил с именем `POLICYA` от двух суперклассов, `X` и `Y`, то они наследуются в отдельности как `X.POLICYA` и `Y.POLICYA`. Случай нечетких правил (fuzzy rule) немного отличается, поскольку нечеткие правила не могут противоречить друг другу, как это объяснено в приложении А. Унаследованные множественные нечеткие наборы правил с одним и тем же именем могут быть объединены. Тем не менее и в четких, и в нечетких случаях именовать наборы правил следует с осторожностью.

Иногда, в простых случаях, можно определить правила, которым нужно подчиняться в соответствии со всеми стратегиями управления для множественного наследования. В случае, когда объекты идентифицируются только абстрактными типами данных (т.е. условные типы, представляющие, например, отношения, не разрешаются), для наследования имеется четкий набор из трех правил.

1. Не должно быть циклов вида: `x есть АКО y есть АКО z есть АКО x`. Это правило исключает избыточные объекты.
2. Основание связи АКО должно быть подтипом, и ее вершина должна быть подтипом или абстрактным типом.
3. Должна быть возможность именования подтипа двух супертипов. Это правило предотвращает наличие абсурдных объектов, например, таких как класс всех людей, которые являются игрушками.

Читателю рекомендуется проверить эти правила самостоятельно.

Цепочки правил

Системы, основанные на правилах (rule-based system), являются непроцедурными, т.е. порядок следования правил не затрагивает их интерпретацию. Правила могут быть сгруппированы в наборы правил, обеспечивающие вывод значений для объекта. В некоторых средах, таких как среда представления и использования знаний KEE (Knowledge Engineering Environment), правила являются истинными объектами, но этот подход не относится к предмету обсуждения. Большинство оболочек и сред экспертных систем предполагают описание правил с последующей идентификацией объектов, используемых правилами. Это предполагает проектирование сверху вниз и может быть полезно для обучения, но противоречит объектно-ориентированному подходу.

Правила могут относиться к нескольким различным типам. Например, есть триггеры, бизнес-правила (business rule) и управляющие правила. Бизнес-правила обычно связывают несколько атрибутов, а триггеры связывают атрибуты с операциями.

```
Бизнес-правило: If Service_length > 5 then Holiday=25
Триггер: When Salary + Salarylncrement > 35000
              run AwardCompanyCar
```

Первое из двух простых приведенных выше правил интересно, поскольку его можно реализовать двумя различными способами. Можно поместить предусловие в метод `getHoliday`, который всегда проверяет переменную `Service_length` перед возвращением значения. В качестве альтернативы постусловие можно поместить в метод `putService_length`, который определяет, должна ли ежегодно изменяться величина `Holiday`. Ясно, что первое соответствует отложенному вычислению, а второе — "энергичному". Проектные решения такого рода *не* должны приниматься во время определения технических требований или анализа. Использование подхода, основанного на правилах, откладывает эти решения до более подходящего момента. Довольно сложные правила могут быть выражены просто как наборы правил. Например, класс `InsuranceSalesmen` мог бы содержать правила для предоставления наилучшего совета заказчику (см. фрагмент кода ниже). Правила срабатывают, когда объекту `BestProduct` требуется значение. Следует обратить внимание на то, что эти правила не

нарушают инкапсуляцию объекта `Client`, устанавливая в нем значение `RiskAverse`. Продавец просто делает предположение в случае отсутствия данных или запрашивает у объекта `Client` эту информацию. Если для `RiskAverse` атрибут `Client.preference` уже установлен, то эти правила никогда не срабатывают. Следует обратить внимание также на непроцедурный характер этого набора правил. Правило, которое срабатывает первым, написано последним. Набор правил для поиска значения `BestProduct` выполняется в режиме обратной цепочки. Таким образом, можно увидеть, что используемый язык является непроцедурным, т.е. упорядочение набора правил не влияет на их интерпретацию.

```
Regime = 'Backward';

Goal = bestProduct;

If client.status is 'Retired'
     and client.preference is not 'RiskAverse'
     then bestProduct is 'Annuity';

If client.status is 'Young'
     and client.preference is not 'RiskAverse'
     then bestProduct: is 'Endowment';

If client.preference is 'RiskAverse'
     then bestProduct is 'Bonds';

If client.children: > 0
     then client.preference is 'RiskAverse';
```

Язык UML и правила

Спецификаторы (qualifier) языка UML могут быть выражены как правила. В качестве примера можно рассмотреть ассоциации "многие ко многим" между файлами и каталогами DOS. Спецификатор `Filename` (имя_файла) сводит это к ассоциации "многие к одному", как проиллюстрировано на рис. 6.20. Вообще, явная квалификация только расчленяет множества. Дело в том, что квалификация относительна; имеется степень уточнения. Во избежание путаницы в объекте `Directory` инкапсулируют правила вида: "Каждый файл в атрибуте `ListOfFiles` должен иметь уникальное имя". Если `FileNames` является атрибутом, этого можно избежать, написав `FileNames` [*множество* имен], а не [*множество с повторяющимися элементами* ...] или [*список* ...].

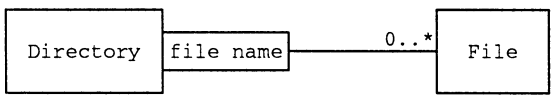


Рис. 6.20. Квалифицированная ассоциация

В языке UML инварианты и наборы правил можно записать в (дополнительном) четвертом именованном разделе, расположенном под разделом операций в изображении типа.

Основанные на правилах расширения объектно-ориентированного анализа помогают обогатить семантику моделей обычных коммерческих систем. Обогащенная семантика делает

эти модели удобочитаемыми и в большей степени обратимыми; большее количество замыслов аналитика становится в модели более наглядным. Это обеспечивает подлинно объектно-ориентированный подход к спецификации расширенных систем баз данных и систем, основанных на использовании знаний.

Как уже было показано, в приложениях содержатся не только правила управления. Уже упоминались бизнес-правила. В обоих случаях инкапсуляция правил в объектах имеет смысл и увеличивает возможность многократного использования и расширяемость. Правила системного уровня подходят для большинства общих объектов в системе, которые являются вершиной иерархии (или иерархий, если отсутствует единый базовый класс, такой, например, как `Object` в Smalltalk). Они распространяются на другие части системы через наследование. Все виды правил сохраняются в наборах правил дополнительного четвертого раздела в изображении объекта.

Поиск

Рассмотрим простой, но забавный пример, включающий правила управления для множественного наследования и значений атрибутов по умолчанию. Предположим, что у автора возникло желание создать старомодную, текстовую приключенческую игру, связанную с построением модели Громадной Пещеры, которую он назвал Поиском. Обычно имеются следующие виды объектов: объекты местоположения (location), исполнители (actor) и сущности (thing). Игра имеет объекты местоположения, в которых могут находиться подвижные объекты (назовем их сущностями, чтобы избежать путаницы) с перечисленными в них атрибутами или свойствами, описанными с помощью операций. Атрибуты объектов местоположения также описывают различные входы и выходы. Имеются игроки, а компьютер имитирует проводника, который будет описывать текущее местоположение игрока, и, если, например, игрок просит его съесть змею, то он может ответить так: "Я только что потерял аппетит". Этих исполнителей удобнее рассматривать как людей, соответствующих классификационной структуре, показанной на рис. 6.21. Следует обратить внимание на то, что проводнику не разрешено подбирать объекты (обозначенные `x`), но он наследует способность передвигаться с места на место. Игроки могут давать команды, такие, например, как "возьми драгоценный камень", "иди на запад" или "ударь тролля ведром".

Теперь рассмотрим структуру объекта `Thing`. Сущности могут быть классифицированы как сокровища, значение которых по умолчанию равно 10 очкам, как оружие, которое не имеет значения по умолчанию, и как сервисные предметы, такие как продовольствие, чемоданы и т.п. Структура условной классификации сущностей показана на рис. 6.22, на котором может быть отмечено злоупотребление множественным наследованием, поскольку инкрустированный драгоценными камнями меч является и сокровищем и оружием. Сокровище имеет положительное значение в очках, а оружие — нулевое. Так что же является значением инкрустированного драгоценными камнями меча, который может наследоваться как от оружия, так и от сокровища?

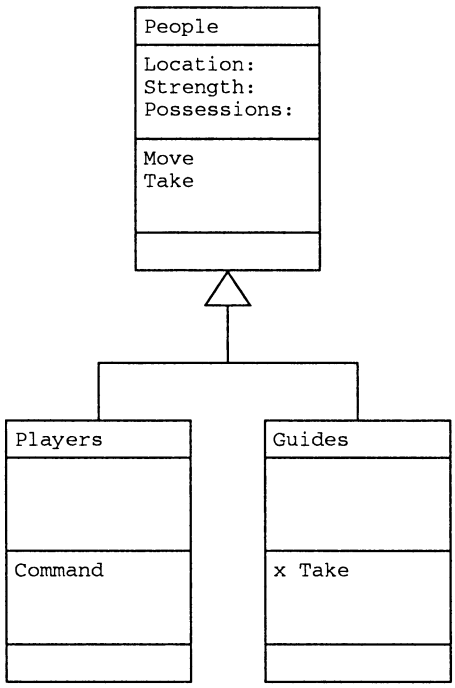


Рис. 6.21. Структура классификации исполнителей игры Поиск

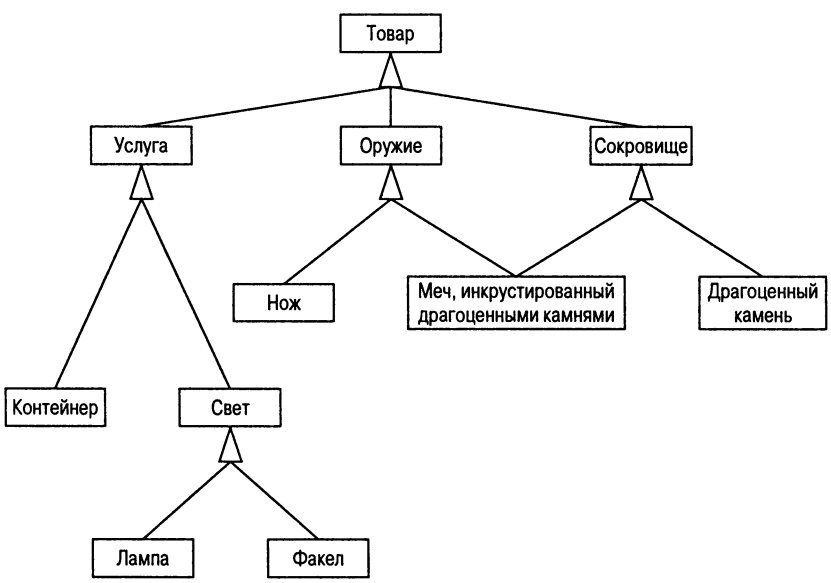


Рис. 6.22. Структура классификации сущностей

Существует несколько стратегий для обработки такого типа конфликтов наследования. Система может сообщить о конфликте пользователю и попросить указать значение, и это является самой обычной стратегией. К сожалению, остановка выполнения программы — это обычно больше чем просто раздражающий фактор; она является дорогостоящей. Все другие подходы требуют учета в системе семантики приложения. В [761] предложены различные стратегии поиска кратчайших путей, основанные на идее, что самое прямое наследование представляет самое конкретное знание. Это может быть полезно, но предположение не всегда обоснованно, и стратегия терпит неудачу, если длины путей равны. В работе [396] предполагается, что конфликт подразумевает незнание и означает необходимость наследовать "неизвестное" `null`-значение. Проблема, связанная с этим подходом, состоит в том, что он воспроизводит все логические проблемы, которые возникают в базах данных из-за `null`-значения, но в некоторых приложениях этот подход имеет смысл. Он может также привести к ошибкам во время выполнения программы. Если это возможно, производному объекту нужно предоставить ответ заранее в соответствии с заданным по умолчанию правилом управления, например не перекрывать унаследованными значениями имеющиеся ресурсы. Наконец, в случае атрибутов, но не операций, унаследованные значения иногда могут быть объединены. Можно, например, взять среднее или взвешенное среднее двух унаследованных числовых значений. В [485] представлен обзор работ в этой общей области интерпретации множественного наследования, получившей название "терминологической логики" (terminological logic).

В приложении А дается образец абсолютно отличной схемы для наследования значений такого типа. В этом приложении также показано, как, используя методы искусственного интеллекта и теории нечетких множеств, можно реализовать частичное наследование (наследуя нечто в некоторой степени, что немного похоже на наследование певческих способностей матери) или частично наследовать свойства (аналогично тому, как стать немного богаче). Но все еще не решена проблема меча, инкрустированного драгоценными камнями. В этом случае хорошим практическим советом является применение для класса `Thing` такого правила: "Если конфликты касаются значения `Value`, то атрибут объединяет значения, принимая наибольшее из них" (рис. 6.23). Меч, инкрустированный драгоценными камнями, наследует это правило. Это является гарантией, что меч получает очки и все еще может уничтожить тролля.

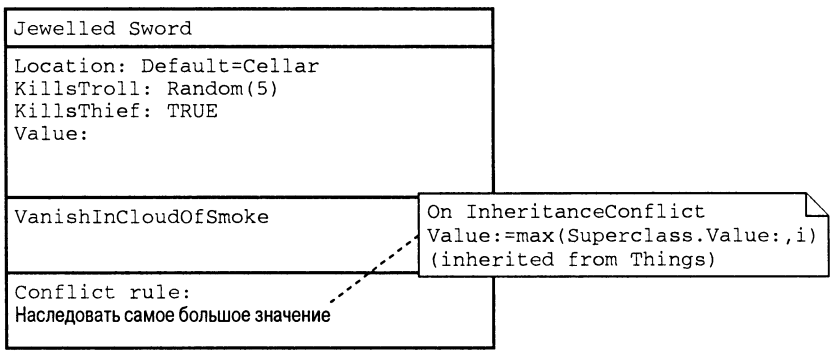


Рис. 6.23. Включение правила разрешения конфликта

Некоторые исследователи применили эту основанную на правилах идею для программирования броузеров. Эти исследователи полагают, что разработчики знают о своих программах многое, что не может быть выражено формально, и что поиск неявных связей ведет к

пустой трате времени. Они предлагают связать код с описательным уровнем, в котором могут быть применены поиск и управление. Например, исследовательская система CogBrow поддерживает такие операции, как: "Найти все подпрограммы, написанные на прошлой неделе Дж. Смитом, и заменить каждое вхождение `GOTO 32767` на `GOTO END:`". Подобными утверждениями и правилами можно манипулировать на описательном уровне. Также были попытки применения объектов с правилами к моделированию объектно-ориентированных сетей [53].

В 1989 году автор при использовании метода Коада-Йордана (Coad-Yourdon) был первым, кто мотивировал необходимость добавить к этому методу правила и не описывать компьютерную систему, а построить модель организации. Этот вид деятельности, известный как обратное проектирование бизнес-процессов (business process re-engineering), выдвинул на передний план потребность в записи бизнес-правил, часто выражаемых поверхностно или расплывчато, как принято у разработчиков систем, основанных на использовании знаний.

Система SACIS

Для более конкретного и реалистичного примера можно исследовать некоторые объекты с правилами, используемые в системе SACIS. Сценарий включает конкретный вид события: продажу. Продажа включает покупателя, который может быть взрослым или ребенком, и перечень товаров различного типа. Есть автомат для считывания штрих-кода, который читает и модифицирует данные из системы складского учета и системы гарантии приемлемости товара для целей покупателя. Термин "приемлемость" охватывает различные критерии "пригодности", одним из которых является безопасность. Предположим, что конкретный покупатель пытается приобрести тюбик клея. Следует обратить внимание на то, что в системе имеется большое количество структур множественного наследования; в частности, одна из них (для клея) показана на рис. 6.24.

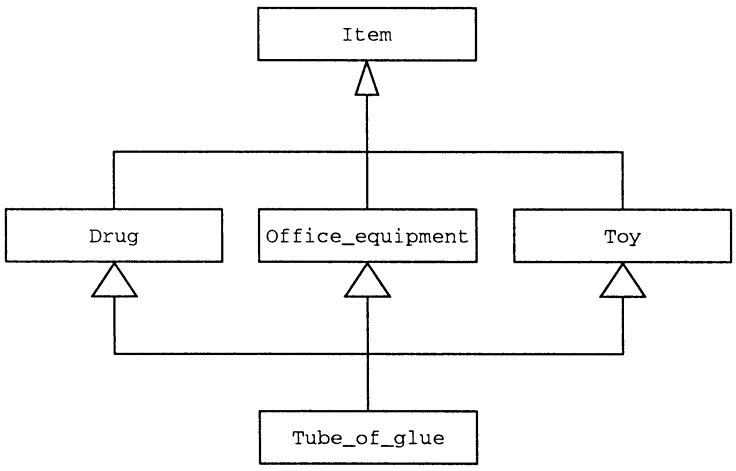


Рис. 6.24. Структура множественного наследования в системе SACIS

Вопрос заключается в том, как моделировать вывод о безопасности этого тюбика клея.

Поскольку клей наследуется от канцелярских товаров и игрушек, с безопасностью не должно быть проблем. Однако клей может быть использован неправильно, как одурманивающий

наркотик. По этой причине теперь в большинстве магазинов отказываются продавать клей детям. Структурное моделирование этого факта представляет одно из бизнес-правил для таких магазинов. Однако, возможно, имеется *реальная* причина для покупки ребенком клея, например была куплена модель самолета, которая должна быть склеена. В таком случае система должна либо разрешить продажу, либо, возможно более строго, отказать в продаже клея или модели (которая является бесполезной без клея).

Чтобы это реализовать, система должна пометить безопасность клея более низким значением и просмотреть всю торговую операцию с целью обнаружения покупок, связанных с клеем. На рис. 6.25 показаны два объекта, которые должны быть выделены во время анализа этой проблемы для преобразования схемы модели данных в объектно-ориентированную форму.

Sales	Tube_of_glue
Customer: item: price: ...	AKO: Toy, Drug, Office_equipment price: size: safety:
readBarCode updateStock	checkOtherPurchases applyRegulations
Правила: продажа является правомерной, если удовлетворяет всем инструкциям	Правила: при наследовании свойств безопасности принимать наименьшее значение, если клиентом является ребенок, а свойство безопасности имеет низкое значение, то выполнить метод checkOtherPurchases

Рис. 6.25. Некоторые объекты модели системы SACIS

6.3.4. Инварианты и инкапсуляция

Как уже указывалось, циклические ассоциации содержат в себе инварианты, а двунаправленные ассоциации нарушают принцип инкапсуляции и поэтому несовместимы с объектной ориентацией. Объектно-ориентированный принцип инкапсуляции подразумевает необходимость отказа при объектно-ориентированном моделировании от двунаправленных ассоциаций в пользу однонаправленных указателей или реальных классов.

В формализме автора, как уже было видно, объекты наряду с обычными атрибутами и методами инкапсулируют наборы правил. В этом разделе несколько глубже рассматривается понятие ассоциаций и их "инверсий" и обращается внимание на некоторые противоречия в терминологии, принятой в среде специалистов по базам данных. Затем автор строит новый базис и показывает, как правила целостности ссылок могут встраиваться в классы, а не прилагаться к внешним ассоциациям, как это обычно делается. Наконец, автор утверждает, что классы с наборами правил способны моделировать любые семантические ограничения целостности. Автор верит, что это имеет значение для проектирования объектно-ориентированных средств на основе CASE-технологии в такой же степени, как и для проектирования баз данных. Благодаря этому доводу можно понять, что для поддержки принципа инкапсуляции

объекты *должны* иметь наборы правил. Наборы правил не являются необязательным дополнением. Автор доказывает, что отсутствие двунаправленных ассоциаций *требует* наличия инкапсулированных наборов правил для представления целостности отношений. Материал этого раздела, по сравнению с оставшейся частью книги, носит немного технический характер, и читатель при первом чтении может его пропустить и перейти к разделу 6.3.5. Однако автор полагает, что сделанные выводы имеют большое значение и лежат в основе теории объектного моделирования.

Большинство популярных методов объектно-ориентированного анализа первого поколения (например, [172, 674, 707]) и несколько других, менее используемых методов предлагают конструкцию, которая устанавливает связь между типами объектов, отличную от обобщения и композиции. Обычно эту конструкцию называют ассоциацией. Некоторые авторы в качестве синонима используют термин "отношение"; в то время как другие все еще используют "отношение" как абстракцию высокого уровня, которая объединяет ассоциацию, зависимость использования, агрегацию, коллекцию, различные особенности наследования и т.д. Такие ассоциации описывают один аспект связи между типами объектов. Этот подход знаком большинству разработчиков, которые применяли методы "сущность-связь". Подход семантически идентичен этому методу, поскольку эти ассоциации касаются только структур данных объектов, а не их поведения (за исключением тех редких экземпляров, где ассоциации используются для обозначения передачи сообщений; например, в объектно-ориентированном методе SSADM [665]).

Изучение литературы по семантическому моделированию данных показало, что имеется два фундаментальных способа связывания структур данных или типов сущностей: это конструкторы и указатели. В [222] объясняется, что при первом подходе акцент делается на построении структур с использованием таких конструкторов, как кортеж или множество. Во втором ударение делается на связывание типов при помощи атрибутов. В 1980-х годах первый подход был доминирующим в значительной степени из-за популярности и широкого распространения реляционных баз данных. В моделях "сущность-связь" имеется два логических типа связей: "отношения сущностей" (entity-relationship) и "отношения отношений" (relationship-relationship). Оба типа представляются множествами кортежей, и никакие связи между ними не сохраняются; СУБД сама должна отыскивать связи. Попытки улучшения реляционной модели быстро привели к системам с большим количеством типов отношений. Эта неудовлетворительная ситуация скоро привела к предложениям о замене этих произвольных систем типов отдельным понятием классов, напоминающих об объектно-ориентированном программировании (см. главу 5 для краткого знакомства с историей этих разработок). Подход, базирующийся на указателях, наиболее естественен для специалистов по объектно-ориентированной технологии, но можно также предположить, что популярность таких методов, как OMT, основывалась в значительной степени на том факте, что разработчики, имеющие подготовку в области реляционных систем, признали эти методы привычными и безболезненными. Опасность здесь заключается в игнорировании объектно-ориентированных принципов и создании моделей, имеющих в значительной степени реляционный характер, вместо истинно объектно-ориентированных моделей.

Ассоциации, используемые в качестве типов

Автор считает, что ассоциации должны быть только однонаправленными, так как иначе нарушается принцип инкапсуляции. Некоторые специалисты по объектно-ориентированной методологии привели доводы против позиции автора, говоря, что всегда можно добавить

атрибуты к ассоциации и создать тип объектов ассоциации. Однако этот новый тип объекта должен хранить ссылки на своих прародителей. Поэтому если принять это в качестве решения, то новые связи должны, в свою очередь, привести к созданию новых типов объектов ассоциации. В какой-то момент будет необходимо остановить этот процесс, но все же потребуется представить остальные ассоциации, связывающие созданные объекты. Это не отрицает полезность типов объекта ассоциации. Некоторые ассоциации настолько важны, что они являются самостоятельными концепциями, т.е. типами объектов. Примером такого понятия являются брачные отношения между людьми.

Разумеется, в каждом конкретном случае можно найти подходящий момент для остановки этого процесса. Создание типов из ассоциаций допустимо только в тех случаях, когда сама ассоциация обладает содержательными свойствами. Таким образом, невозможно преобразовать *все* ассоциации в типы, хотя посредством этого процесса обычно удаляют все ассоциации "многие ко многим". Стоит отметить, что если в реляционных базах данных запрещено использование отношений "многие ко многим", то в современных объектно-ориентированных базах данных их использование допускается. Таким образом, привычка удаления таких отношений во всех случаях (обычная среди разработчиков баз данных) теперь не обязательно является хорошей практикой.

Ассоциации и отображения

Приступим к решению упомянутых выше проблем. Для их преодоления аннулируются двунаправленные ассоциации в пользу однонаправленных ассоциаций или **отображений** (mapping), которые можно рассматривать в качестве указателей, встроенных в типы объектов, связанных с наборами правил для сохранения ссылочной и семантической целостности. Последний пункт является наиболее значимым. Инкапсуляция инвариантов класса в объектах обеспечивает хранение правил целостности ссылок.

Класс может инкапсулировать общедоступные наборы правил, которые относятся к другим общедоступным характеристикам класса. Они обеспечивают информацию второго порядка для этого класса. Например, простое правило может быть представлено триггером базы данных, инициирующим при изменении значения атрибута выполнение метода, или оно может представлять необходимую последовательность методов. Более необычные правила могут описывать режимы управления, в соответствии с которыми работает этот класс. Например, в правилах можно устанавливать приоритет значений по умолчанию при разрешении конфликтов множественного наследования. Наборы правил для разрешения конфликтов множественного наследования, касающиеся числовых атрибутов, могут включать математические формулы. Например, можно использовать формулу вычисления средней величины для двух конфликтующих наследуемых атрибутов. Наборы правил — это неупорядоченные множества продукционных правил, которые всегда подчиняются определенным правилам логического вывода. Это означает, что классы могут логически определять факты, которые не были описаны явно.

В некоторых существующих объектно-ориентированных методах ассоциации рассматриваются как отображения. В методе MOSES [378] пропагандируются однонаправленные ассоциации или отображения, а в методе SOMA их называют просто "ассоциациями" или "атрибутами ассоциаций". Во втором методе некоторые ассоциации определяются как "чистые атрибуты" в том случае, если они указывают на примитивные типы. Это позволяет некоторым инструментальным средствам автоматически создавать диаграммы упорядоченных ассоциаций. В действительности на таких диаграммах зачастую представляют только

неориентированные связи ("ассоциации"), но на основной модели, которая детально описывает реализацию, нужно отображать уточненные однонаправленные связи. Некоторые специалисты в области методов разработки отмечают, что ассоциации могут быть представлены в виде отображений, но сами до сих пор работают непосредственно с ассоциациями и не могут объяснить, каким образом при использовании отображений может быть представлена целостность.

Из однонаправленных ассоциаций при необходимости можно создавать типы объектов. Если aMan состоит в браке с aWoman, тогда оба (и aMan, и aWoman) должны знать об этом. С точки зрения этих двух индивидуумов этого достаточно. Однако для регистрации фактов рождения, смерти и брака важен сам объект регистрации брака Marriage; этот объект имеет такие атрибуты, как дата, время и место, и даже может демонстрировать поведение, например может вычислять стоимость юбилейного подарка. В этом случае он действительно является типом объекта. Но он *не* является отображением объекта aMan на объект aWoman. Его природа более независима. Тип объекта регистрации брака должен отвечать за идентичность партнеров, а партнеры должны хранить однонаправленные связи (отображения) с экземпляром регистрации брака. Это не нарушает инкапсуляции, так как и Man и Woman отображаются на класс Marriage. Это не *тип отношения* (второго или даже первого класса), а истинный *тип объекта.*

В [522] высказывается мнение, что ассоциация является парой "отображений", соответствующих *ролям* этой ассоциации [47]. Отображение — это однонаправленная связь. Оно является частью объекта (клиентского) и поэтому не нарушает инкапсуляции. Ассоциация *в стиле* языка UML или методов Коада (Coad), Румбаха (Rumbaugh) или Шлеер-Меллора (Shlaer/Mellor) является внешней и действительно нарушает инкапсуляцию. В данном разделе рассматриваются приводимые Оделлом (Odell) аргументы и представляется их более строгое обоснование.

Ограничения целостности

При замене двунаправленных ассоциаций отображениями возникает очевидная проблема, заключающаяся в отсутствии явного контейнера для хранения ограничений целостности ссылок. Поэтому покажем, как эту проблему можно решить, используя отображения. К счастью, наборы правил обеспечивают четкое решение.

Чтобы рассматривать "ассоциации" как пары ролей или отображений, необходимо хранить отношения между этими парами отображений (если они имеются). Это важно в тех случаях, когда имеются правила целостности. Возвратимся к примеру с браком, но забудем об объекте регистрации брака. Объект aMan состоит в браке с объектом aWoman, и aWoman состоит в браке с aMan. Если aMan не состоит в браке (разводится) с aWoman, это является необходимым условием того (во всяком случае в большинстве стран), что aWoman также не состоит в браке с aMan. Но у объекта aMan нет сведений о знаниях объекта aWoman. Поэтому aMan и aWoman должны хранить наборы правил, которые инкапсулируют "бизнес-правило", или, по крайней мере, триггер-запрос к некоторой внешней базе правил. Отсутствие наборов правил в языке UML препятствует применению этой возможности, хотя можно было бы с тем же эффектом жестко запрограммировать некоторые методы. Разумеется, такие наборы правил не всегда являются необходимыми. Например, aMan и aWoman могут помнить номера телефонов друг друга, и aWoman может забыть номер aMan без воздействия на сопряженное отношение.

Инверсии

Базируясь на терминологии, сложившейся в работах по базам данных и, особенно, по функциональным базам данных [338], Оделл предложил пары связанных отображений с противоположными направлениями считать взаимообратными (в смысле теории множеств). Это на самом деле некорректно, но достаточно близко к правде. Автор полагает, что взаимосвязанные пары однонаправленных ассоциаций (или, более точно, отображений, как они будут называться с этого времени) являются сопряженными функторами (adjoint functor) в смысле теории категорий [540]. Эта идея не будет анализироваться далее в этом разделе, поскольку утверждение о сопряженности не влияет на аргументы автора, хотя он допускает, что будет правильнее использовать термин *сопряженность* (adjoint), а не *инверсия* (inverse).

В математике отношение — это множество упорядоченных n-кортежей, а отображение — это однозначное отношение. Рассмотрим два типа объектов A и B. Забудем, что они являются типами, и рассмотрим их просто как множества. Строго говоря, отображение преобразовывает элементы A в элементы B. Однако такая интерпретация не включает необходимого в данном случае понятия ассоциации, поскольку она не может представить отношения "один ко многим", например "имеет детей"для множеств всех людей A и B. Следуя совету Абриала (Abrial), Оделл (Odell), в соответствии с теорией по функциональным моделям данных, утверждает, что ассоциация состоит из двух взаимообратных отображений [522]. Это соображение не является достаточным для того, чтобы закрыть дальнейшее исследование. Чтобы это увидеть, следует рассмотреть двунаправленную ассоциацию публикаций, представленную на рис. 6.26.

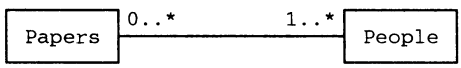


Рис. 6.26. Двунаправленная ассоциация

Эту ассоциацию можно разбить на два отображения.

$$f\text{: } Papers \rightarrow \Psi(People)\ (f' = writtenBy);$$
$$g\text{: } People \rightarrow \Psi(Papers)\ (g' = wrote).$$

Здесь $\Psi(A)$ представляет показательное множество A: множество всех подмножеств A. Эти функции не являются обратными друг для друга.

$f'.g=1$ (g' — это правая инверсия f)
$g'.f=1$ (g' — это левая инверсия f)

Корректнее сказать, что отображения должны выполняться между показательными множествами, так что

$$f\text{: } \Psi(Papers) \rightarrow \Psi(People);$$
$$g\text{: } \Psi(People) \rightarrow \Psi(Papers).$$

Эти отображения полностью определяются их значениями в одноэлементных подмножествах, поэтому, для всех подмножеств S и T, $f(S \cup T)=f(S) \cup f(T)$. В этом смысле эти отображения всегда могут быть "реконструированы" из отображений, области определения которых являются множествами, а не показательными множествами, например $f(\{p\})=f'(p)\ \forall\ p$.

Следует отметить, что для любого множества A $\Psi(A)$ является полной решеткой с отношением предпорядка, определенного включением множеств $\supseteq$ ("содержит"). На основе этого автор утверждает, что эти два отображения являются скорее сопряженными, нежели взаимообратными.

Выбор двух произвольных противоположно направленных отображений между парой показательных типов не является достаточным для истинной ассоциации. Интуитивно понятно, что между элементами этой пары должны существовать некоторые отношения. Это определение дает возможность представить надлежащим образом отношения вида "многие ко многим" и комбинировать эти два отображения, как требовалось первоначально. Но это тоже не обеспечит взаимообратных отношений, как того требует Оделл. Можно лишь утверждать, что эти отображения будут являться **полу-инверсными** слева и справа (где знак = в определении инверсии заменен на знак $\supseteq$):

$$\text{Правило 1: } f.g(\{p\}) \supseteq \{p\}.$$

Здесь $f.g(\{p\})$ — это множество всех людей, которые написали (или написали в соавторстве) статьи, написанные p; оно содержит $\{p\}$.

$$\text{Правило 2: } g.f(\{q\}) \supseteq \{q\}$$

Здесь $g.f(\{q\})$ — это множество всех статей, написанных людьми, которые написали q; оно содержит $\{q\}$.

Интуитивно ясно, что эти два правила представляют минимальное ограничение целостности ссылок для пары отображений, необходимое для того, чтобы составить из них ассоциацию. Фактически **ассоциация** определяется, как пара противоположных отображений показательных типов, которые удовлетворяют приведенным выше правилам 1 и 2. Ассоциация не должна быть слишком строгой, чтобы требовать целостности ссылок, и в разделе А.2.3 автор покажет, как для достижения этого нужно усилить данные условия.

Правила целостности и инкапсуляция

Два упомянутых выше условия могут быть выражены наборами правил (или, более определенно, инвариантами класса), инкапсулированными в классах следующим образом.

Каждый тип объекта соответствует набору классов, которые его реализуют. В качестве примера можно выбрать один из классов. Отображение `f` — это атрибут экземпляра класса `Papers`, который может быть описан как `writtenBy(People,1,n)`. Этот указатель на класс `People` показывает, что статья может быть написана одним человеком или многими людьми (авторами). Тогда в этом примере `g` — это атрибут экземпляра `People`, который обычно описывается как `wrote(Papers,0,n)`.

Поскольку ассоциации являются атрибутами, которые формируют часть интерфейса, экземпляр `q` класса `Papers` может определить множество экземпляров класса `People`, которые написали эту статью. Назовем это множество идентификаторов `AuthorsOf(q)`. Для каждого члена этого множества, экземпляры класса `People` могут проверить значение атрибута `Wrote` и возвратить множество статей, написанных этим автором. Назовем это множество `WrittenBy(AuthorsOf(q))`. Приведенное выше правило 2 только утверждает, что `q` является членом этого множества (для всех `q` из класса `Papers`). Правило инкапсулировано в классе `Papers`. Аналогично правило 1 инкапсулировано в классе `People`.

Учитывая очевидную кардинальность отношений, эти атрибуты и правило 2 можно записать и в более привычном стиле.

- Для каждой статьи существует один человек или несколько людей, которые ее написали.
- Для каждой статьи `q`, `q` является элементом множества статей, написанных людьми, которые написали `q`.

Читатели, которым более близки Smalltalk-подобные языки программирования, а не базы данных, могут рассматривать упомянутые выше факты через призму передачи сообщений, а не отображений.

Поскольку `f` и `g` содержатся в открытых интерфейсах (public interfaces) соответствующих классов, при описании правил инкапсуляция не нарушается. Существование отображения подразумевает, что в одном классе хранится информация о другом классе, что, разумеется, обеспечивает доступ к его интерфейсу. Инкапсуляция нарушается в том случае, если класс ссылается на экземпляр другого класса, поскольку это предполагает его инстанцирование. Поэтому предполагается, что ссылка на экземпляр осуществляется через расширение класса (которое является частью его открытого интерфейса).

При использовании этого подхода не теряется информация о целостности и полностью поддерживается инкапсуляция. Другой подход, принятый в таких методах, как OMT [82, 674] или метод Мартина и Оделла [522], где правила являются внешними по отношению к объектам, не обладает этим преимуществом. Инкапсуляция наборов правил повышает эффективность многократного использования не только классов, но и самих правил, поскольку они легко передаются через структуру классификации (с помощью наследования). Наборы правил обеспечивают возможность формирования логического вывода, что делает их более полными и мощными по сравнению с инвариантами класса. Однако в данном примере точно такой же эффект может быть достигнут с помощью инвариантов класса, таких как в методах MOSES [378] или BON [775].

Приведенное выше предположение не позволяет отыскать тот член ассоциации, который стал причиной нарушения целостности (поскольку правило применено *апостериори*), хотя это необходимо для любой СУБД. Эту проблему можно решить следующим образом. Предположение, что второй класс может выполнять свою функцию `g` в обратном направлении (исходя из результата), не является нарушением инкапсуляции. Исходя из этого, на первом этапе необходимо отыскать множество авторов статьи `aPaper`, `wrote(aPaper)`, передав сообщение классу `People`. Чтобы гарантировать целостность, достаточно сравнить образ возвращаемого множества `wrote(aPaper)` со значением `writtenBy` соответствующего экземпляра статьи `aPaper` класса `Papers`. Это рассуждение более подробно разъясняется в следующем разделе.

Правила обеспечения целостности ссылок

Еще один способ рассмотрения правила 1 состоит в том, что `q` содержится в *объединении* (union) отдельных множеств статей, написанных каждым автором `q` (представленных в ассоциации `writtenBy` класса `Papers`). Эти множества находятся в классе `People` с помощью проверки ассоциации `wrote` для каждого автора, который написал статью `q`. По сути, этого недостаточно для того, чтобы гарантировать целостность ссылок.

Предположим, установлено, что статья q находится в ассоциации `writtenBy({Hugh})`. Тогда автор Hugh должен установить, что эта статья q содержится в образе ассоциации `wrote`. Если автор Hugh не может установить этого по какой-либо причине, то по правилу 1 для класса `Papers` нужно сравнить объединение множеств статей, написанных авторами `Authors`, которые написали q в соответствии с информацией класса `People`. Целостность будет нарушена в том случае, если статья q написана в соавторстве с другими авторами и в результирующем множестве содержится q, хотя ассоциация `wrote` автора Hugh не содержит статью q. Таким образом, данное правило не дает возможности отыскать тот член ассоциации, который стал причиной нарушения целостности, хотя такой поиск необходим для любой СУБД.

Решением является построение *пересечения* (intersection) (а не объединения) всех приложений отображения. Если пересечение (множеств статей, написанных каждым автором) не содержит статью q, то можно констатировать нарушение целостности. Таким образом, классы `People` и `Papers` должны содержать следующие правила.

Правило 3: $\{p\} \subseteq \bigcap_{q_a \in g(\{p\})} f(\{q_a\})$

Правило 4: $\{q\} \subseteq \bigcap_{p_a \in f(\{q\})} g(\{p_a\})$

Инкапсуляция этих правил в классах, связываемых ассоциацией, гарантирует целостность ссылок и придает точный смысл интуитивному понятию "инверсных" отображений. Однако справедливо следующее утверждение.

Утверждение 1. Правила 3 и 4 обеспечивают более слабые условия, чем высказывание, что f и g являются взаимообратными.

Доказательство.

Предположим, что g является левой обратной функцией для f, так что $gf(Q)=Q$. Выберем любое q, и пусть $pa \in f(\{q\})$. Тогда $g(\{pa\}) \subseteq gf(\{q\})=\{q\}$. Поэтому $g(\{pa\})$ или пусто или равно $\{q\}$. Полученное в результате пустое множество нарушает приведенные ранее предположения и подтверждает правило 4 (замена включения на более сильное равенство).

Обратное утверждение неверно. Это можно доказать, рассматривая группу из трех авторов: Hugh, Ivan и Jane, которые являются соавторами двух статей q и r. Здесь $\{q\}$ — это строгое подмножество множества $\{q,r\} = g(f(\{q\})) \cap g(f(\{r\}))$. На рис. 6.27 проиллюстрирован этот контрпример.

Доказательство закончено.

Чтобы получить истинные инверсии, необходимо в этих двух правилах использовать равенства, но это несколько не очевидно. **Строгую ассоциацию** необходимо определить как пару отображений, удовлетворяющих правилам 3 и 4. Подобную пару можно также определить как пару **строгих полуинверсий**.

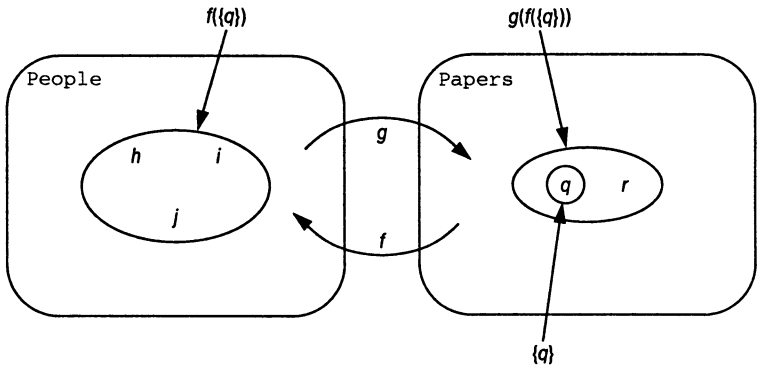


Рис. 6.27. Контрпример из доказательства утверждения 1

Семантическая целостность

Давно известно, как представляются ограничения целостности ссылок в сетевых, реляционных и объектно-ориентированных базах данных. Однако общий метод представления ограничений семантической целостности, т.е. наиболее общих ограничений, накладываемых на связи сущностей или объектов, ускользнул от коллективного осмысления специалистами в области баз данных. Некоторые успехи были достигнуты в отношении дедуктивных баз данных, вследствие чего стало понятно, каким образом можно представить такие ограничения, как "все дети, которые любят конфеты, также любят и собак". Но этот результат не был обобщен в отношении всех ограничений или других типов баз данных, особенно объектно-ориентированных. Наиболее общий метод представления ограничений семантической целостности состоит в том, чтобы использовать триггеры баз данных, но известно, что он охватывает не все возможные семантические ограничения. В этой книге автор трактует данную проблему как проблему моделирования или представления знаний и предлагает общее решение, которое может быть легко реализовано в пределах объектно-ориентированных баз данных и, с небольшой доработкой, в других базах данных.

В этом подразделе показано, что для моделирования целостности ссылок с использованием однонаправленных отображений необходимым и достаточным условием является применение инвариантов класса. Однако эти доводы были представлены с использованием наборов правил (ruleset). Широко известно, что наборы правил (порождающие правила), работающие в непроцедурном режиме, например, обратной цепочки (backward chaining), соответствуют завершенной машине Тьюринга. Таким образом, им доступны все вычисления или (что эквивалентно) им могут быть представлены все утверждения о семантике некоторого объекта изучения. На этом основании автор утверждает, что объекты, инкапсулирующие наборы правил, обеспечивают очень удобный способ представления в объектно-ориентированном стиле самых общих ограничений семантической целостности.

Остается проблема реализации этих общих моделей в произвольной СУБД, но сама модель является общей и компактной. Для реализации модели могут быть применены средства процедурного языка и/или триггеры баз данных: какие-либо общие рекомендации по программированию дать невозможно. При процедурной реализации теряется трассируемость абстрактной модели. Кроме того, в модели может обеспечиваться полная инкапсуляция и поддерживаться возможность повторного использования спецификации, но в реализации это

может не сохраниться. В главе 5 уже рассматривалось, каким образом некоторые идеи автора, особенно касающиеся инверсий и ссылочной целостности, уже реализованы в типичной современной объектно-ориентированной системе управления базой данных.

6.3.5. Модели состояний

Для объектов со сложными состояниями полезно использовать диаграммы состояний, принятые в языке UML, или диаграммы переходов STD (State Transition Diagram). Эти диаграммы используются для описания динамики отдельных объектов и связаны с объектной моделью как средство эффективной спецификации операций и формальных утверждений, описывающих их семантические свойства. Диаграммы состояний (statechart) представляют возможные последовательности изменений состояния. Каждый переход — это реализация какого-либо действия или прецедента. Во время определения технических требований эти модели нужно строить очень внимательно с учетом специфики предметной области. По мнению автора, эта методика не всегда подходит для объектов, используемых в информационных системах управления или MIS-системах (Management Information System), и главные выгоды этой методики проявляются во время физического проектирования. С другой стороны, эта методика является привычной и полезной для многих специалистов, работающих в области телекоммуникаций.

Еще одна проблема, касающаяся чрезмерно активного использования диаграмм переходов, состоит в том, что они могут очень быстро усложняться. Однако для некоторых объектов со сложными жизненными циклами данная методика неоценима для описания системной информации и бизнес-процессов. Хорошим примером системы, для которой полезна эта методика, является приложение для оформления заявок на получение ссуды. Однако сам факт представления этого бизнес-процесса в качестве класса вряд ли заслуживает внимания. Часто этот вид информации лучше описывать в модели прецедентов, на основе которой затем строятся такие классы, как класс ссуд. Представление бизнес-процессов будет рассматриваться в главе 7.

Значимые состояния объекта определены значениями его атрибутов. Эти состояния и допустимые переходы между ними фиксируются на диаграммах переходов. На рис. 6.28 иллюстрируется основная система обозначений, применяемая при построении таких диаграмм. Состояния, которые могут описываться булевыми атрибутами или значениями конкретного "атрибута состояния", представлены прямоугольниками с закругленными углами с указанием имен. Переходы — это помеченные стрелки, соединяющие состояния.

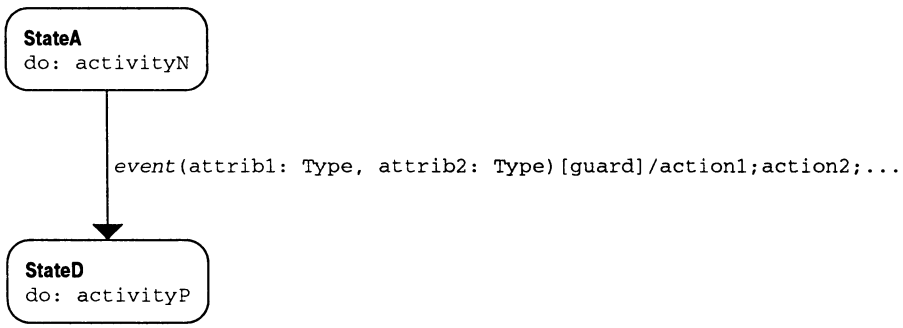


Рис. 6.28. Состояния и переходы

События, которые становятся причиной переходов, описаны рядом со стрелками. Эти события тоже могут содержать атрибуты. Например, можно представить себе в качестве объекта калькулятор, событие "нажатие клавиши" которого может иметь атрибут `key_number`. В этом случае отсутствует необходимость в отдельных событиях для каждой клавиши. С событиями могут связываться предусловия — логические условия, которые должны быть истинны для выполнения перехода. Действие (action) — это функция, которая выполняется немедленно после того, как происходит переход. С состояниями могут быть связаны виды деятельности (activity) — функции, которые начинают выполняться сразу же после входа в состояние и продолжают выполняться до своего полного завершения или до выхода из этого состояния, в зависимости от того, что наступит раньше. В предусловии можно потребовать, чтобы до завершения вида деятельности данное состояние не изменялось.

Некоторые специалисты, в том числе Хендерсон-Селлерс (Henderson-Sellers) (в частной беседе), обращают внимание на то, что новички часто неправильно используют виды деятельности и даже злоупотребляют ими, применяя их для отображения потоков данных на диаграммах переходов. Поэтому использование видов деятельности не поощряется.

На рис. 6.29 дан простой пример, отражающий часть жизненного цикла всплывающего меню (pop-up menu). В этом примере начальное и конечное состояния изображены в виде соответственно закрашенного круга и круга с внешним ободком. Точки `Начало` и `Конец` (здесь `Выбрано`) не являются состояниями, а лишь представляют создание и разрушение экземпляров, вследствие чего атрибуты становятся неопределенными.

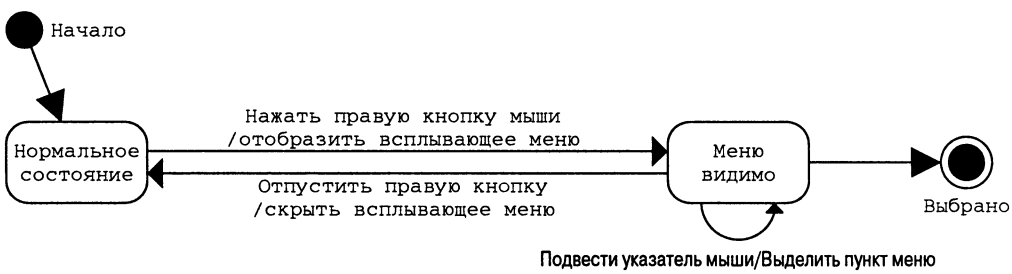


Рис. 6.29. Пример

Для более четкого представления структуры можно определять вложенные и параллельные состояния. На рис. 6.30 представлена большая часть системы обозначений, принятой для описания диаграмм состояний. Сюда, во избежание путаницы, не включены некоторые детали, например начальное и конечное состояния, которые не очень важны для понимания основных идей этой системы обозначений.

Подтипы наследуют модели состояний своих супертипов, как показано на рис. 6.31, где изображены диаграммы для простого графического редактора.

Иногда состояния типа полезно представлять в качестве *настоящих* типов, например вместо того чтобы делить людей на работающих и безработных, можно создать типы для служащих и т.д. Их можно считать ролями, как рассматривалось выше, или **типами состояния** (stative type). Последнее представление полезно в том случае, когда атрибуты и поведение очень различаются.

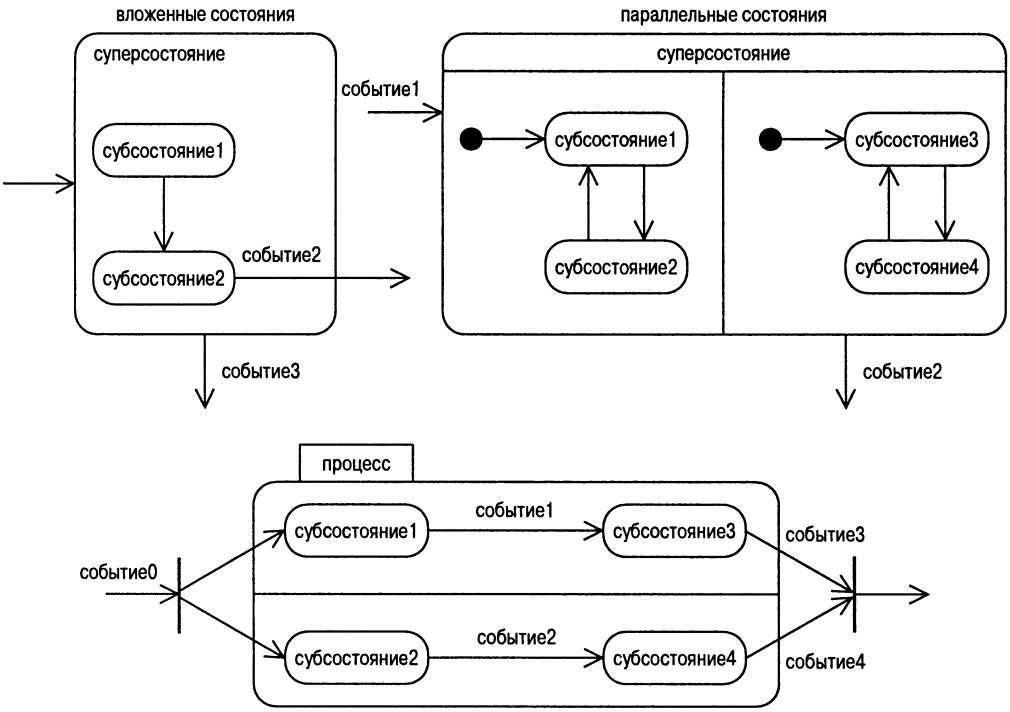


Рис. 6.30. Система обозначений, принятая в диаграммах переходов (STD)

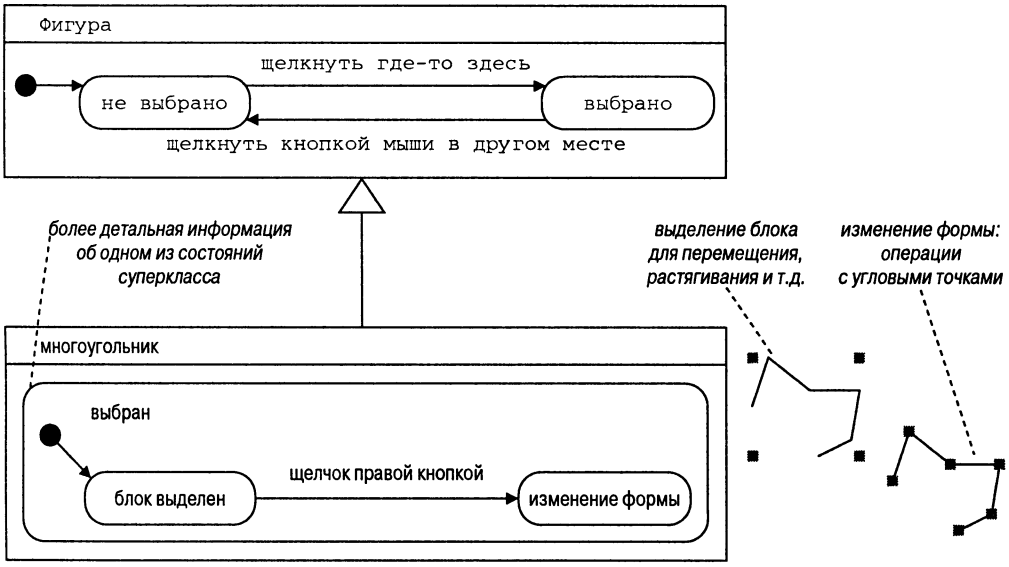


Рис. 6.31. Состояния и подтипы

Диаграммы состояний, наряду с диаграммами последовательностей, являются полезными для исследования и документирования деталей прецедентов и выявления способа реализации конкретного бизнес-процесса.

Диаграммы состояний описывают локальную динамику каждого объекта в системе. Однако остается необходимость в методах, позволяющих описывать глобальное поведение. Использование наборов правил помогает локально инкапсулировать некоторые аспекты глобального поведения. Диаграммы состояний обеспечивают дополнительный способ визуализации событий и обычно позволяют выявлять прецеденты, которые были пропущены при первом просмотре. Поэтому они важны для обратной связи с прецедентами. Диаграммы состояний проясняют, какие последовательности действий являются допустимыми.

6.3.6. Переход к компонентному проектированию

В процессе разработки системы спецификация преобразуется в проектное решение и набор компонентов, которые составляют эту систему, поэтому каждое состояние необходимо описать более детально. Диаграммы последовательностей и сотрудничества концентрируют внимание на прецедентах в пределах системы. Как правило, к ним добавляются временные ограничения, предусловия, зависимости между объектами (такие, как `uses` (использует)) и определяется, являются ли сообщения последовательными или асинхронными. В первом случае изменение состояния необходимо подвергать блокировке до получения ответа или нужно использовать некий временной период ожидания.

Типы, содержащиеся в спецификации на рис. 6.32, представляют модель, а не приказ для конструктора, реализующего проектный замысел. Любой проект приемлем, если он порождает поведение, подразумеваемое спецификациями прецедентов. Используемые в модели термины описываются в словаре. Необходимой частью спецификации являются только прецеденты, показанные в нижней части рисунка. Типам могут быть назначены операции, но они интерпретируются как "факторизованная" спецификация. Если для класса `Guest` определена операция `reallocateRoom`, то система должна содержать некоторое средство с этим именем, одним из параметров которого является `Guest`. Ничего не говорится о том, как это средство должно быть реализовано.

Выборки

Спецификация напоминает этикетку на коробке, в которой находится система. Она описывает, каким образом будет вести себя система. В ней указываются обязанности, которые определяются прецедентами, и их цели. В спецификации также описывается словарь, который может использоваться для описания этих обязанностей, т.е. для создания модели типов. С другой стороны, в спецификации не даются формулировки обязанностей отдельных типов в пределах модели, за исключением ассоциаций. В методе Catalysis такие спецификации иллюстрируются способом, показанным на рис. 6.32. Модуль компонента может иметь несколько таких интерфейсов, соответствующих различным наборам прецедентов и исполнителей. Показанная на рис. 6.32 этикетка (label) фронтальной грани содержит часть интерфейса, имеющего отношение к регистрации гостя. Могут быть другие этикетки, имеющие отношение, скажем, к обслуживанию комнат, их распределению или оплате персонала.

В спецификации на самом деле не говорится, что находится внутри коробки. Объекты на этикетке могут быть простой иллюзией, созданной в соответствии с шаблоном Facade (фасад). Для описания полного интерфейса обычно необходима концептуальная модель внутреннего состояния, которая позволяет строить диаграммы состояний и последовательностей.

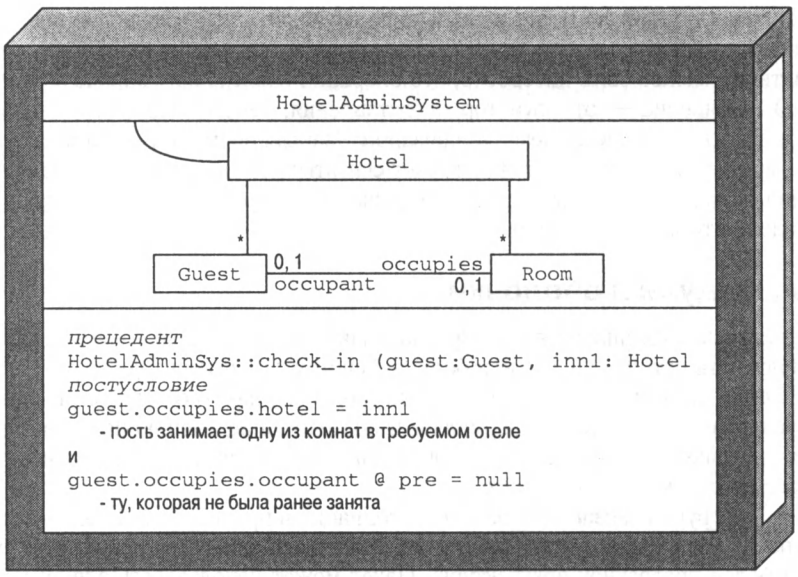


Рис. 6.32. Спецификация метода
Catalysis = модель типов + результаты прецедентов

В методе Catalysis представлена важная идея **извлечения** (retrieval), заимствованная из формальной спецификации языков. Выборка — это функция, которая определяет значение абстрактного атрибута из фактической реализации. Она показывает, каким образом атрибуты соответствуют абстракции (если эти две модели находятся в соответствии друг с другом, т.е. имеется отображение спецификации в проект системы с обоснованием принятых проектных решений). Таким образом, хотя реальная модель и не требуется, но возможность ее извлечения (retrievable) должна существовать. В качестве примера можно предположить, что имеется спецификация очереди следующего вида.

Queue
length: Integer ...
put (*post*: length = length@pre + 1 and ...) get (*post*: length = length@pre - 1 and ...)

Теперь предположим, что имеются две реализации этой очереди на основе массивов. В одной реализации переменная `len` содержит длину, и ее значение увеличивается или уменьшается на единицу каждый раз, когда элемент добавляется к очереди (queue) или извлекается из нее с помощью методов `put` (добавить) и `get` (получить). Тогда (очевидно) функция извлечения: `length = len`. Во второй реализации поддерживаются два указателя `in` и `out`, которые указывают на элементы массива, куда может добавляться и откуда, соответственно, может удаляться новый элемент. Теперь функция извлечения `length = (out - in) mod n`, где `n` — размерность массива.

Такая операция извлечения дает возможность проверить постусловия во время выполнения в режиме отладки.

Математикам может быть интересно, что операция извлечения сопряжена с уточнением (refinement). Уточнение — это функтор представления, действующий от спецификации до реализации, в то время как извлечение связано с "забыванием". Это интересно, поскольку наводит на мысль, что ни один метод не может считаться законченным без *обоих* этих понятий; причина в том, что теоремы сопряжения (зачастую теоремы представления) находятся в центре каждой математической теории.

Пакеты, модули и оболочки

Обычно классы объединяются в дискретные пакеты. Любой язык моделирования, позволяющий описывать нетривиальные или большие системы, для уменьшения сложности моделей описываемых систем должен включать некоторый вид пакетной стратегии. Большинство методов предполагает довольно неформальный подход к определению пакетов. Обычно при формировании пакетов руководствуются эвристикой, предписывающей хранить тесно связанные классы в одном и том же пакете.

Пакет — это группа связанных классов, ассоциаций и прецедентов. Пакетирование обеспечивает способ организации больших моделей, описания различных предметных областей и реализации более прозрачной архитектуры. Пакет может использовать элементы (зависеть от них), определенные в других пакетах, поэтому он не является абсолютно самостоятельной единицей. Предпочтительно, чтобы такие зависимости использования между пакетами были нециклическими. Это дает возможность при импортировании одного пакета не импортировать все остальные. На рис. 6.33 показаны обозначения для пакетов, принятые в языке UML, а также примеры удачных и неудачных архитектур. Большинство инструментальных средств может автоматически изображать отношения зависимости между пакетами, и чаще всего пакеты можно экспортировать и импортировать из одной модели в другую. Наиболее очевидной аналогией между обозначением пакета в UML и средствами его реализации на конкретном языке является понятие пакета в Java. **Модельные шаблоны** (model template) метода Catalysis являются параметризованными пакетами.

На диаграммах модулей или компонентов UML используются "градиграммы" (Gradygram), которые уже упоминались в главе 1. Компоненты могут соответствовать узлам (node) — физическим компьютерам. Узлы изображаются в виде параллелепипедов, подобных тому, который показан на рис. 6.32. Если компонент имеет несколько интерфейсов, каждый из них изображается в виде "леденца на палочке", но это более подробно будет рассмотрено в главе 9 при обсуждении компонентно-ориентированной разработки как способа объектно-ориентированного проектирования. Что касается обозначений UML для диаграмм видов деятельности (activity diagram), то они будут рассматриваться в главе 7.

Иногда полезно иметь более строгое понятие о том, что находится в пакете. Автор чувствует необходимость в более строгом определении инкапсуляции (т.е. сокрытия) данных в пакете. Первоначально в методе SOMA это понятие именовалось как уровни (layer). Но в настоящее время ясно, что это слово должно быть зарезервировано для представления уровней (клиент/сервер) приложений, связанных через порты программного интерфейса API (Application Programming Interface), как определено в семиуровневой модели стандарта OSI или методе ROOM [694]. Поэтому теперь инкапсулирующие единицы пакетирования называются **оболочками** (wrapper). Оболочки инкапсулируют свои компоненты. Нет необходимости в новом обозначении для оболочки, поскольку это просто составной объект (composite

object), для которого можно использовать стандартное обозначение композиции, принятое в языке UML. Так как оболочки — это компоненты, а спецификации компонентов — это типы, то для них можно использовать обозначения компонентов или типов языка UML. К тому же, как в предыдущей главе, можно придать термину "оболочка" дополнительное значение, чтобы он означал также объект, инкапсулирующий обычную традиционную систему. Это не является насилием по отношению к языку, поскольку простой унифицированный объект — это не что иное, как несложная традиционная система, состоящая из функций и данных.

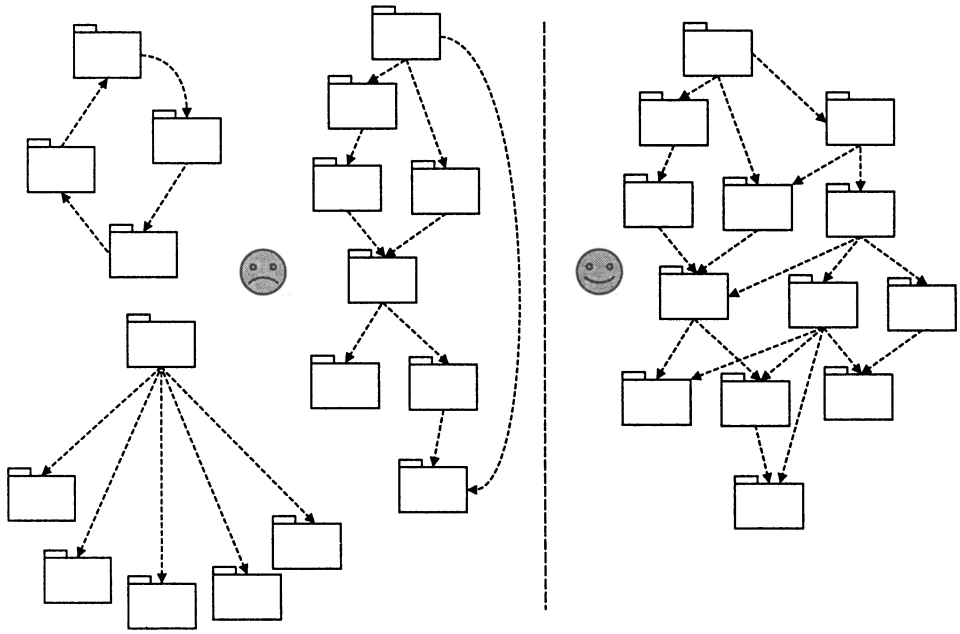


Рис. 6.33. Зависимости между пакетами

Оболочки делегируют некоторые или все обязанности, описанные в их интерфейсах, операциям соответствующих компонентов. Это делегирование относится к связям вида `implemented-by` (реализуется посредством). Непосредственно в UML они еще не поддерживаются, но в ожидании этого можно комментировать операцию с помощью следующего синтаксиса.

```
/* Implemented-by    Classname.operation_name */
```

Это представление соответствует идее делегирования оболочками некоторых или всех своих обязанностей операциям их компонентов. Что касается системы обозначений, то уровни можно было бы также показывать с использованием градиграмм, как на рис. 6.34. Это обозначение иногда предпочтительнее, поскольку так проще показывать связи вида `implemented-by`. Следует обратить внимание на то, что входящие сообщения не могут проникнуть через оболочку, а исходящие могут.

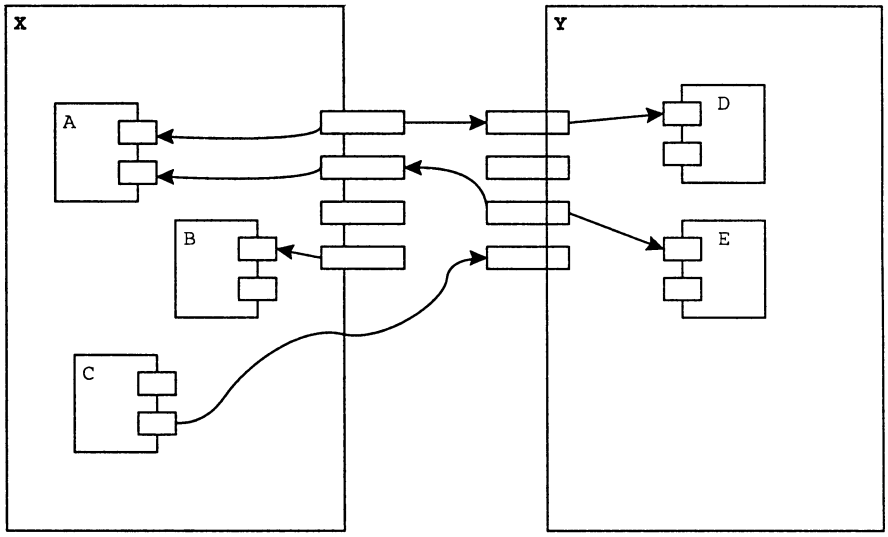


Рис. 6.34. Пакетирование и делегирование

Оболочки инкапсулируют бизнес-функции и облегчают использование существующих компонентов, которые не обязательно являются объектно-ориентированными. Типичным приложением оболочек является прикладной уровень обычной трехуровневой архитектуры.

Жизнеспособной альтернативой использованной здесь неформальной системе обозначений является обозначение **кластера** (cluster) из метода OML [272], которое здесь не приводится, поскольку может увести слишком далеко от основной темы. Можно также использовать обозначения для **пакета**, принятые в языке UML. Более подробная сравнительная характеристика этих двух обозначений оболочки содержится в [598].

6.3.7. ШАБЛОНЫ ИЛИ МОДЕЛЬНЫЕ КАРКАСЫ

Объекты взаимодействуют с другими объектами, и этими взаимодействиями определяются роли объектов. Если объект играет несколько ролей, то все они воздействуют на его состояние. Например, если кто-то является и служащим и родителем, то одна роль пополняет счет в банке, а другая — его опустошает. С этой точки зрения взаимодействия являются фрагментами многократного использования в такой же степени, как и сами объекты.

Взаимодействия всегда были в центре объектно-ориентированного проектирования. Распределение обязанностей среди объектов — это ключ к успешной автономизации (decoupling). Фактически в большинстве шаблонов проектирования отношения между несколькими объектами описываются с указанием конкретных ролей, например Subject-Observer (Субъект-Наблюдатель) [291]. В методе Catalysis представлен один из способов моделирования таких шаблонов сотрудничества на основе идеи модельного каркаса [791]. "Модельный каркас" и "модельный шаблон" являются синонимами и соответствуют понятию сотрудничества (collaboration), принятому в языке UML 1.3 (не путать с термином "сотрудничество" (collaboration), описанным в [282]).

Модельный каркас (model framework) — это связанная группа пакетов, являющаяся основой для разработчика. Она имеет смысл только тогда, когда ее абстрактные концепции

могут быть заменены довольно конкретными сущностями. В методе Catalysis показано, как это делается с помощью понятия "заменяемые заполнители" (substitutable placeholder). Как правило, каркас является шаблоном многократного использования.

На рис. 6.35 предполагается, что сначала была построена модель предметной области, касающейся слесарных работ, а позднее было осознано, что можно абстрагироваться от деталей слесарных работ и рассмотреть более общую проблему распределения ресурсов, согласованную с этим же примером. Именно поэтому создается макроверсия модели. Термины, которые изменяются в зависимости от предметной области, заменены абстрактными "заполнителями". При конкретизации шаблона, "заполнители", в соответствии с правилами переименования, заменяются "реальными" типами, приведенными в затененном прямоугольнике.

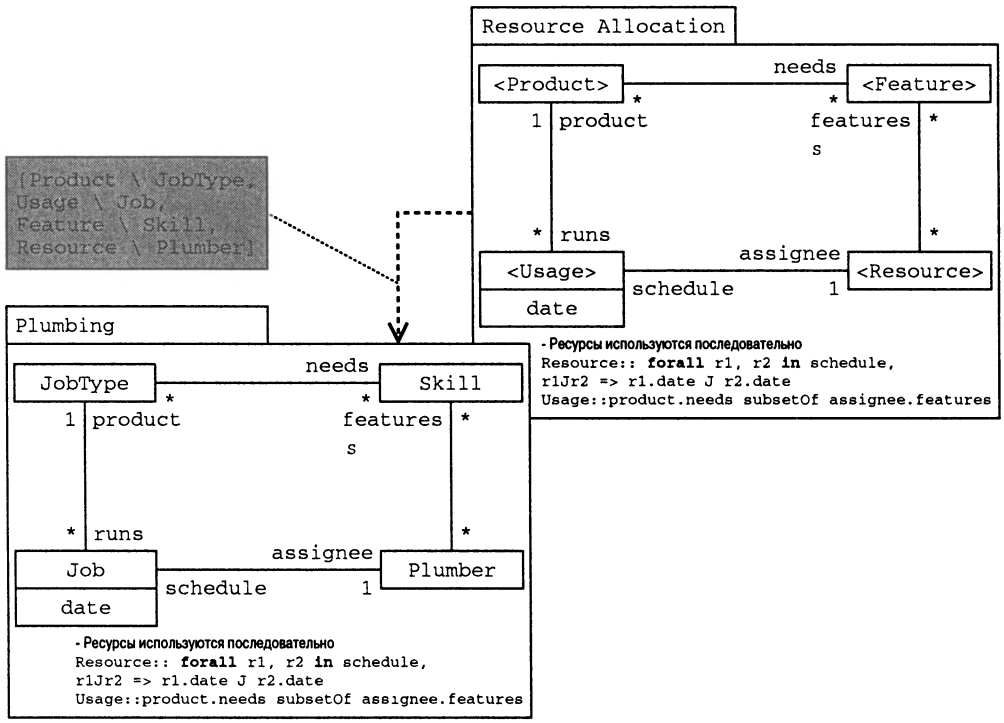


Рис. 6.35. Шаблоны каркасов

Шаблон каркаса, или для краткости шаблон, является, таким образом, пакетом с параметрами вида `<placeholderName>`. Это не то же самое, что наследование, поскольку заполнители не содержат ничего такого, что может быть "унаследовано" в строгом смысле этого слова. Однако инстанцированный класс "наследует" все ассоциации с их кратностью, так что, в некотором смысле, это немного более сильное понятие, чем наследование. Все типы, ассоциации и ограничения порождаются при конкретизации (инстанцировании) этой макроверсии.

В качестве условного обозначения для понятия сотрудничества можно использовать соответствующее обозначение, принятое в языке UML. На рис. 6.36 связи, направленные к типам, показывают применение шаблона `ResourceAllocation` (распределение ресурсов). В шаблоне к этим типам добавляются атрибуты. Эллипс — это не прецедент, а **шаблон** (pattern) сотрудничества, принятый в UML. Применение шаблона добавляет возможность инстанцирования существующих типов.

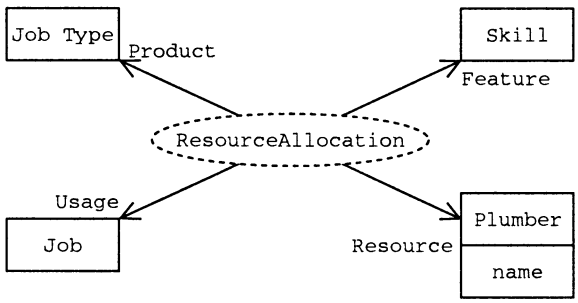


Рис. 6.36. Применение шаблона каркаса

В одном и том же приложении шаблон можно применить более одного раза. На рис. 6.37 показано, как его можно дважды использовать в контексте управления обучением. Заполнителем ресурса в одном случае является `Instructor` (преподаватель), а в другом — `Room` (помещение). В этом примере необходимо скорректировать модель типа не только с помощью явной подстановки типа, но и посредством устранения неоднозначности ассоциаций путем их переименования, как показано на рис. 6.38. Также необходимо уточнить любые неоднозначные инварианты. Таким образом, инвариант, изображенный на рис. 6.35, для объекта `Instructor` принимает такой вид.

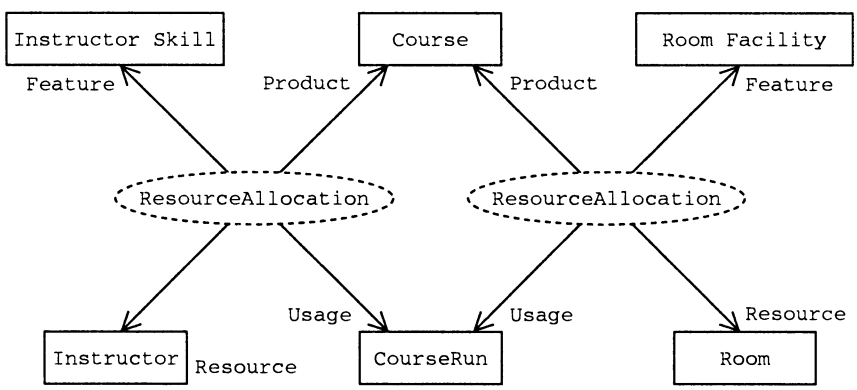


Рис. 6.37. Многократное применение шаблона

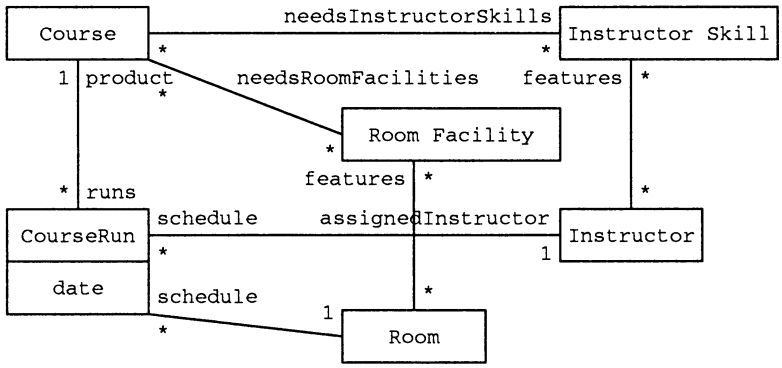


Рис. 6.38. Развернутая модель типов для применения шаблона

```
- Each Instructor has one CourseRun on a given Date
- (Каждый преподаватель ведет один учебный курс в
- данный момент времени)
Instructor:: forall r1,r2 in schedule,
          r1 ? r2 => r1 .date ? r2.date
CourseRun:: product.needsInstructorSkills
          subsetOf assigneeInstructor.features
```

Для объекта Room потребуется совсем другое представление.

Каркасы, так же как и типы и ассоциации, могут включать прецеденты и их постусловия. Они могут быть представлены графически (в виде эллипсов) или в виде словесного описания. Это очень помогает при компонентном проектировании, которое будет рассматриваться в главе 9. Как видно из рис. 6.39, даже простой прецедент Sales (продажи), который рассматривался выше в этой главе, может быть представлен в виде шаблона. На рис. 6.40 детализированы развернутый тип и спецификация прецедента для этого составного объекта. Средства, поддерживающие идею каркасов, позволяют достраивать частичные модели, применяя эти каркасы к классам.

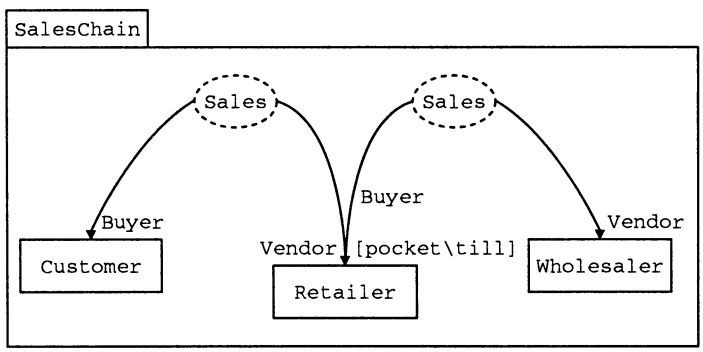


Рис. 6.39. Инстанцирование обобщенной модели

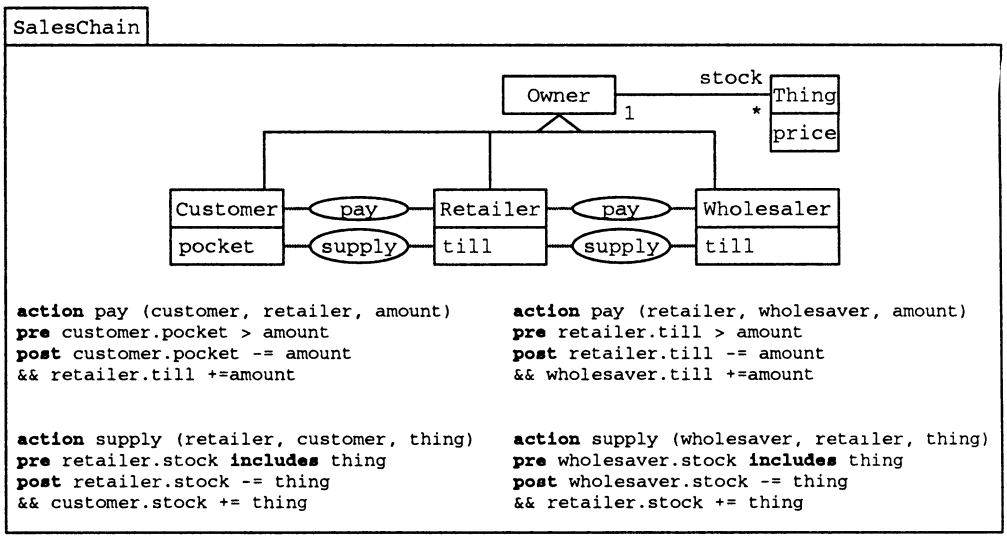


Рис. 6.40. Развернутый составной объект

6.3.8. Процесс проектирования

Метод Catalysis рекомендует для спецификации системы и ее компонентов, а также для проектирования использовать микропроцесс, проиллюстрированный на рис. 6.41. Можно начать либо с действий, либо с объектов (или, возможно, с того и другого). В первом случае выявляются действия, в которых система принимает участие, и выясняются субъекты действия (исполнители). После того как для действий сформулированы постусловия, тщательно пересматривается словарь, необходимый для интерпретации модели типов. Затем для уточнения и усовершенствования модели используется методика моментальных снимков, диаграммы последовательностей и состояний, что ведет к появлению новых дополнительных действий. И так далее по циклу, представляя больше деталей, и, в конечном счете, продвигаясь от спецификации к проектированию и реализации.

Важные принципы проектирования, которые следует применить в рамках этого процесса, сводятся к следующему.

- Распределять обязанности между объектами равномерно, пытаясь получить объекты приблизительно одинакового размера.
- Гарантировать, что все ассоциации являются направленными. Уточнить инварианты, используя моментальные снимки.
- Избегать зацикливания и сильного ветвления на выходе (как показано на рис. 6.33).
- Удостовериться, что зависимости упорядочены в соответствии с уровнями.
- Минимизировать степень связывания и область видимости между объектами.
- Применять шаблоны в процессе проектирования (см. главу 9).

- Производить итерации, пока модель не станет устойчивой или не наступит срок завершения проекта.

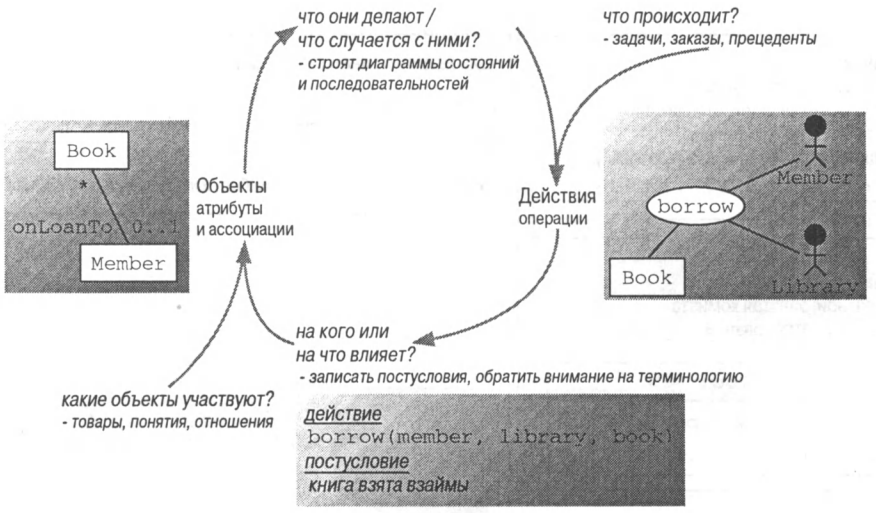


Рис. 6.41. Микропроцесс определения спецификации и проектирования метода Catalysis

6.3.9. Документирование моделей

Одной из больших опасностей, которая неизбежно возникает при покупке CASE-средства, является соблазн изготовления громоздких диаграмм классов с документированием каждого класса и массы дополнительных деталей. Инструментарий такого вида позволяет это сделать без особого труда. Спустя несколько недель результаты такой работы будут полностью бесполезны и, вероятно, неудобочитаемы для любого, за исключением разработчика, который их получил.

Маленькие диаграммы иногда полезны для иллюстрации описательной части документации, но при этом необходимо помнить, что только разработчики разговаривают на языке UML. Не следует ожидать, что пользователи будут способны их интерпретировать, по крайней мере без подсказки. Поэтому текст должен объяснять информацию, отображенную на диаграмме. На рис. 6.42 иллюстрируется одобряемый автором стиль документирования. Диаграммы состояний и последовательностей тоже должны быть в тех местах, где они поясняют текст.

Обычно важным компонентом документации является глоссарий или словарь проекта. Он определяет типы и все связанные с ними атрибуты, операции и инварианты. Он может также включать часть материала по бизнес-процессам, которая соответствует предметной области, но не показана в модели типов. Классам должны быть присвоены ключевые слова для обеспечения возможности их автоматического поиска. Хорошим примером синтаксиса является: `topic = Biology; owner = Smith;` и т.д.

Строго говоря, все диаграммы избыточны, поскольку все они могут быть сгенерированы автоматически из описаний типов [329]. Однако для некоторых читателей иллюстрация может быть полезной и может помочь в уточнении необходимых сведений.

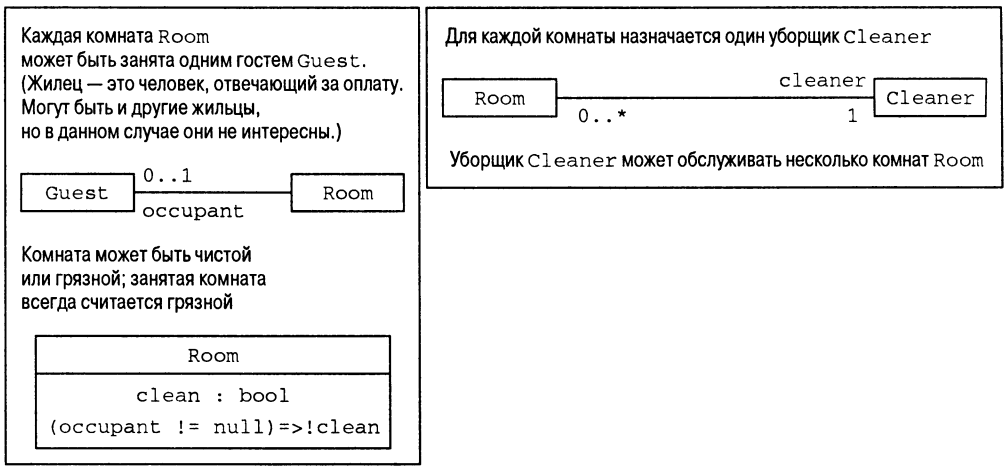


Рис. 6.42. Стиль документирования

6.3.10. Расширения для приложений реального времени

Аплеты, выполняемые в броузерах, и множество приложений реального времени влекут за собой проблемы, связанные с параллелизмом их выполнения. Этот вид программ может (или кажется, что может) делать несколько вещей одновременно. Параллельную последовательность шагов можно представить с помощью потоков (thread), которые могут запускаться, останавливаться и передавать сигналы друг другу. Простым и знакомым примером является изображение в формате GIF (Graphics Interchange Format — формат графического обмена), загружаемое в броузер в то время, как пользователь продолжает щелкать на гиперссылках. Потоки являются ключевой частью объектной модели в языке Java. В [130] описана важная и мощная методика **временных потоков** (time thread). Временные потоки (автор, воспользовавшись удобным моментом, отмечает, что в упомянутой книге они определяются как "карты прецедентов") дают возможность проектировщикам визуализировать способ передачи управления между компонентами системы. В этом смысле поток реализует "внутренность" прецедента. Однако эти авторы обычно путают прецеденты со сценариями, и большинство приведенных ими примеров касается использования временных потоков для моделирования сценариев, а не более общих прецедентов. Карта временного потока содержит компоненты системы (которые могут быть машинами или людьми) и одну или несколько волнистых линий, которые проходят через компоненты и комментируются с помощью (возможно, совместных) обязанностей. В роли компонентов могут выступать объекты или чистые процессы. Временные потоки могут разветвляться и соединяться. Они могут синхронизироваться в соответствии с рядом протоколов. На самом деле можно привести исчерпывающий список возможных шаблонов связывания процессов. Потоки могут обращаться к "пулам" информации и динамически наполнять объектные "слоты"

С помощью диаграмм временных потоков может быть представлен ряд стандартных шаблонов проектирования. Они варьируются от абстрактных и общих шаблонов, подобных Producer/Consumer (производитель/потребитель), до вполне определенных шаблонов реального времени, таких, например, как динамическая буферизация. Это могут быть конкретные шаблоны для проектирования графического интерфейса пользователя, такие как MVC. Система обозначений довольно сложна, и автору кажется, что ее не стоит использовать для систем, которые не включают каких-либо сложных связей между процессами. Тем не менее при сетевом проектировании или при использовании средств действительно низкого уровня изучение этой методики может принести некоторые дивиденды. Хотя для заурядных MIS-систем она слишком сложна. На рис. 6.43 показано, как параллельные потоки могут выглядеть на диаграмме потоков.

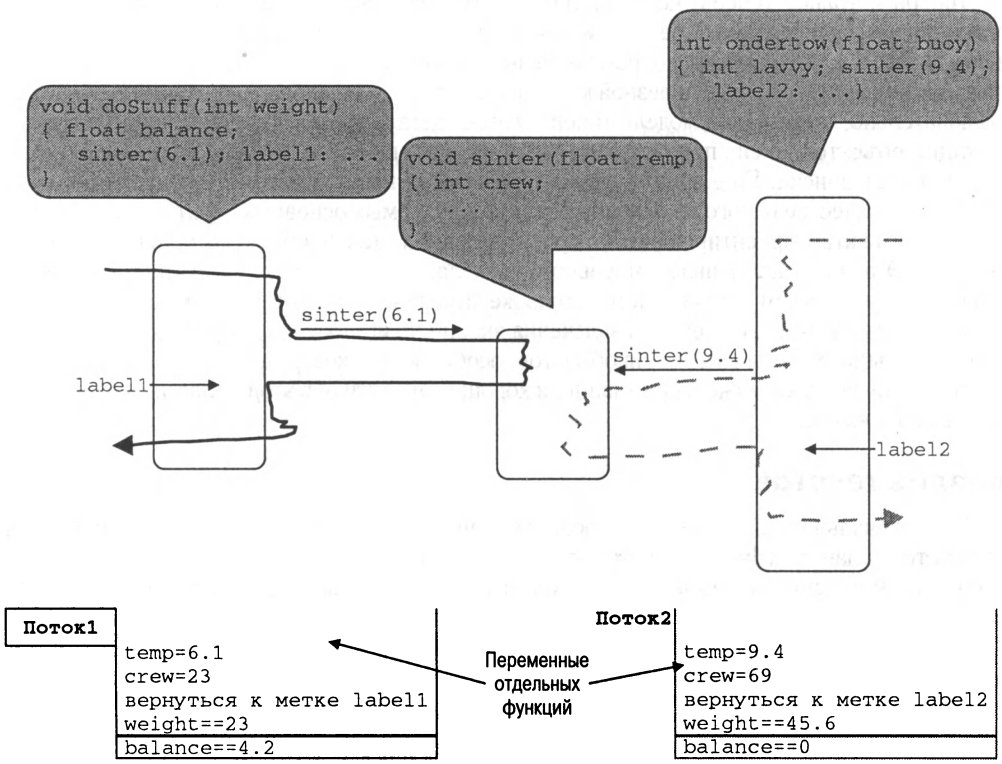


Рис. 6.43. Временные потоки

Данная методика является приемом проектирования высокого уровня, который достигает своего апофеоза на полпути между анализом прецедентов и объектным моделированием. Фактически временные потоки было бы полезно добавить к большинству методов, особенно для тех случаев, где необходимо моделировать параллелизм или поведение системы в реальном времени. Временные потоки также было бы полезно добавить к методам представления сценариев задач, описанным в главе 7. Сценарий основной задачи, связанный с передачей сообщений в модели бизнес-процессов, инициирует временной поток. Этот метод проектирования

более гибок, чем диаграммы последовательностей или видов деятельности, поскольку дает возможность проектировщику задерживать передачу сообщений.

В методе ROOM для объектно-ориентированного проектирования в реальном масштабе времени [694] вводится понятие портов и идея представления их протоколов в качестве классов. Это послужило основой системы обозначений расширения языка UML для задач реального времени. Об этих вопросах будет сказано больше при обсуждении компонентного проектирования в главе 7, в которой также рассматривается идея капсул.

6.4. Идентификация объектов

Теперь и только теперь, когда имеются интеллектуальные средства, необходимые для описания и моделирования объектов, можно начать думать о том, каким образом их выявлять и создавать. Как уже было видно, распознавание существительных и глаголов в постусловиях прецедентов является очень полезной методикой для создания словаря предметной области и, следовательно, статической модели типов. Однако детальное обсуждение процесса идентификации объектов до сих пор откладывалось, поскольку ранее не было возможности описать предмет поиска. Еще одна причина заключается в том, что идентификация объектов признана в качестве одного из основных (возможно, самых основных) критических элементов как объектно-ориентированного проектирования, так и объектно-ориентированного анализа. Эта тема заслуживает отдельного раздела, в котором автор попытается показать, что идентификация объектов является столь же химерическим критическим элементом, как и знаменитый критический элемент извлечения знаний Фейгенбаума (Feigenbaum). Существуют специальные методы выявления объектов, особенно тех, которые уже используются в моделировании данных и инженерии знаний, и хорошие аналитики в этой области уже практически не спотыкаются.

Анализ текстов

Первоначально объектно-ориентированный метод проектирования Буча начинался с анализа потоков данных (dataflow), которые затем использовались для идентификации объектов посредством просмотра и выделения конкретных и абстрактных объектов в пространстве задачи на основе диаграмм потоков данных DFD (Data Flow Diagram). Следующий шаг — это выявление методов с помощью диаграммы DFD. Альтернативный и в то же время дополняющий подход, впервые предложенный в [2], заключается в извлечении объектов и методов из текстового описания задачи. Объекты соответствуют существительным, а методы — глаголам. Глаголы и существительные могут быть далее разделены на подклассы. Например, имеются имена собственные и несобственные, глаголы, выражающие действие (doing), принадлежность (being) и наличие (having). Глаголы, выражающие действие, обычно преобразуются в методы, глаголы, выражающие принадлежность, — в классификационные структуры, а глаголы, выражающие наличие, — в композиционные структуры[7]. Переходные глаголы, как

[7] При этом следует соблюдать осторожность, поскольку глаголы могут отражать состояние. В английском языке эти глаголы являются составными конструкциями, например "to be present" (быть присутствующим) отражает состояние, а "to increase" (увеличивать) — нет. Возможно, более удобным промежуточным языком для объектно-ориентированного проектирования был бы китайский, поскольку в нем четко различаются глаголы, описывающие состояние. Тем более, что он более прост в изучении, чем Ада.

правило, соответствуют методам, а непереходные могут относиться к исключениям или к зависимым от времени событиям, например в такой фразе, как "магазин закрывается". Это является полезным руководящим принципом, но не может рассматриваться в качестве какого-либо формального метода. Для хорошего проектирования требуется интуиция. Эта методика может быть автоматизирована, что и сделано в некоторых средствах, разработанных для поддержки метода HOOD [680].

Например, описание технических требований к системе SACIS может содержать следующий фрагмент.

> ... Если **покупатель** *входит* в **магазин** с намерением *купить* игрушку для **ребенка,** то **совет** относительно пригодности игрушки для ребенка должен *быть предложен* в пределах разумного времени. Он зависит от **возраста** ребенка и **атрибутов** игрушки. Если игрушка **относится к опасным предметам,** то она является непригодной ...

Автор жирным шрифтом выделил некоторые потенциальные (candidate) классы (существительные), а курсивом — некоторые потенциальные методы (переходные глаголы). Возможные атрибуты или ассоциации подчеркнуты. Процесс может быть продолжен с учетом рекомендаций, изложенных в табл. 6.1, но необходимо понимать, что это **не является** формальной методикой в общем смысле этого слова. Таблица предназначена просто для стимуляции мыслительного процесса в период анализа.

Большинство методов, основанных на работе Буча (Booch), включая методы, базирующиеся на языке UML, используют анализ текстов подобного типа.

В методах HOOD, RDD и некоторых других используется анализ текстов Эбботта (Abbott), а в остальных методах проектирования строгие рекомендации относительно выявления объектов вообще отсутствуют. В [172] указано, что аналитики должны выявлять запоминаемые объекты или события, устройства, роли, узлы и организационные единицы. В методе Шлеер-Меллора (Shlaer/Mellor) выделяется пять категорий: материальные сущности (tangible entity), роли (role), эпизоды (incident), взаимодействия (interaction) и спецификации (specification). Это очень хорошо, но не точно и, безусловно, не полно. В [168] утверждается, что существует ровно 5 базовых видов объектов, и приводятся свойства каждого из них. Каждому базовому типу приписывается цвет (как бы цвет этикетки). Перечень базовых типов сводится к следующему.

- описания (синий)
- участники (например, люди, организации), места или предметы (зеленый)
- роли (желтый)
- моменты или интервалы времени (розовый)
- интерфейсы и точки подключения (см. главу 9)

Трудно поверить, что мир действительно настолько прост, но идея является полезной и подчеркивает важность шаблонов анализа, которые будут рассматриваться в главе 9. В этом разделе предоставляется несколько довольно точных, нормативных методик выявления объектов (табл. 6.1). Они основаны на инженерии знаний, а также практике и философии человеко-машинного взаимодействия HCI[8].

[8] Human-Computer Interaction — человеко-машинное взаимодействие.

Таблица 6.1. Рекомендации по анализу текстов

Часть речи	Компонент модели	Пример
имя собственное	экземпляр	Дж. Смит
имя несобственное	класс/тип/роль	игрушка
глагол, выражающий действие	действие	купить
глагол, выражающий принадлежность	классификация	является
глагол, выражающий наличие	композиция	имеет
глагол, отражающий состояние	инвариантное условие	принадлежат
модальный глагол	семантика данных, предусловие, постусловие или инвариантное условие	должен быть
имя прилагательное	значение атрибута или класс	непригоден
фраза, использованная в качестве прилагательного	ассоциация	покупатель с детьми
	действие	покупатель бумажного змея
переходный глагол	действие	входить
непереходный глагол	исключение или событие	зависеть

6.4.1. Философия познания и теория классификации

Для понимания подходов к идентификации объектов полезно сделать паузу и рассмотреть некоторые из существующих философских представлений этой темы. В течение очень долгого времени доминирующим философским видом научного мышления Запада был эмпиризм. Автор полагает, что этот вид представления скрывает сущность объектов и не должен использоваться объектно-ориентированным аналитиком. Например, эмпиризм рассматривает объекты в качестве простых связок свойств. Как уже упоминалось в главе 1, эмпирик утверждает, что объекты находятся где-то рядом в ожидании их восприятия, и в экстремальном варианте эмпиризма считается, что это делается очень просто. Феноменологическое представление, напротив, признает, что восприятие является активным, итерационным, творческим процессом и что объекты появляются как из наших представлений, так и из объективно существующего мира в строго повторяющемся диалектическом процессе. Понятие дерева (до его восприятия) как такового вызывает представление о деревьях вообще. Как выразился известный биолог Д.З. Янг [815], когда мы смотрим, "уже знаем, что собираемся увидеть". В более богатом представлении предполагается, что абстракции реального мира являются отражением социальных отношений и процессов, а также наличия реального базиса. Когда мы наслаждаемся красотой дерева, воспринимаем дерево в качестве посредника между нашими собственными человеческими процессами производства деревянных изделий и нами самими. В работах [250, 800] эти проблемы исследуются в общих чертах относительно компьютерных систем.

Объекты, которые могут быть постигнуты аналитиком, знакомым со множеством различных способов их применения и выбирающим "лучшее" представление, являются фундаментальным вкладом человеческого интеллекта в компьютерные системы. Некоторые объекты непосредственно соответствуют реальным объектам, например людям или стульям, а другие, такие как стеки или кварки, — вымышленным абстракциям. Абстракции обобщают соответствующие понятия предметной области. Абстрактный тип может зависеть от приложения, но в этом случае будет поставлена под угрозу возможность его повторного использования, поэтому проектировщик должен попытаться выделить наиболее универсальные абстракции. С другой стороны, должны быть предприняты меры для того, чтобы избежать снижения эффективности из-за чрезмерно обобщенных объектов. Чем выше уровень системы, тем в большей степени она должна рассматриваться как средство улучшения общения между людьми. Чем большее значение придается социальной стороне проблемы, тем в большей степени при идентификации объектов на передний план выдвигается субъективный фактор.

Как полагает автор, большинство объектно-ориентированных методов анализа не помогают идентифицировать объекты. Достаточно разумно заниматься анализом существительных, глаголов и других частей речи в неформальном описании задачи или набора постусловий прецедента. При этом используются следующие эмпирические правила. Имена собственные ставятся в соответствие экземплярам, а несобственные имена существительные — типам или атрибутам; употребленные в качестве прилагательных фразы, квалифицирующие существительные, такие как "служащий, который работает в отделе зарплат", определяют отношения или методы, если они содержат глаголы, например "служащий, который получил повышение".

Методика Эбботта (Abbott) полезна, но сама по себе не может быть успешной. Этот слабоструктурированный подход является только руководящим принципом, и, как уже подчеркивалось, необходимо творческое осмысление проблемы опытными аналитиками. Использование этой методики затруднено. Аналитики и проектировщики могут многое почерпнуть из двух ключевых разделов философии: эпистемологии и онтологии. Эпистемология — это теория познания. Она изучает природу знания, его истоки и степень познаваемости мира. Онтология — это наука о жизни, ее сущности и возникновении. Эти две науки тесно связаны, потому что нас интересует истинное знание: знание того, что есть в действительности. Обе дисциплины касаются природы объектов и поэтому соответствуют идентификации объектов при объектно-ориентированном анализе и проектировании. Эта точка зрения была представлена в [773], где предложена формальная модель объектов, основанная на математической онтологии Банджа (Bunge). К сожалению, эта онтология является атомистической; она представляет сущности как сводимые к неделимым компонентам. Тем не менее принцип, основанный на онтологическом подходе, хотя и не на формальной модели, является обоснованным.

Существенные и случайные утверждения

В [119] отмечается различие между сущностью и случайностью в программной инженерии. Это различие является очень старым, идущим от Аристотеля и средневековых схоластов. Понятие сущности подвергается резкой критике современными философами, начиная с Декарта, который рассматривал объекты как простые связки свойств, лишенные внутренней сущности. Это привело к возникновению серьезных трудностей, поскольку в этом случае невозможно объяснить, как распознать стул, не имеющий общих свойств со стульями, с которыми ранее имели дело. Школа познания, известная как феноменология, представленная такими философами, как Гегель (Hegel), Брентано (Brentano), Гуссерль (Husserl) и Хайдеггер

(Heidegger), возникла в результате попыток решения этого вида проблем. Еще одной классической проблемой, важной для объектно-ориентированного анализа, является проблема категорий. Аристотель использовал набор фиксированных пар категорий, применение которых является основой познания. Это были такие понятия, как универсальность/индивидуальность (universal/individual), необходимость/возможность (necessary/contingent) и т.д. Кант (Kant) пересмотрел этот перечень, но, как заметил Гегель, при этом не стал отягощать себя излишними заботами. Идеалист Гегель доказывал, что категории связаны между собой и вытекают друг из друга. Наконец, материалист Маркс (Marx) показал, что мыслительные категории являются результатом социального и исторического опыта человека.

> Мой диалектический метод не только отличается от гегелевского, но и является его прямой противоположностью. По Гегелю, жизненный процесс человеческого мозга, т.е. процесс мышления, который преобразовывает идею в независимый субъект, является творцом реального мира, и реальный мир — это только внешняя, воспринимаемая форма этой "идеи". Что касается меня, то, напротив, идеальное — это не что иное, как материальный мир, отраженный человеческим сознанием и переведенный в мыслительные формы. [528].

Итак, мы не только наследуем категории от наших предков, но и познаем новые категории из своего жизненного опыта.

Все, феноменалисты и диалектики, идеалисты и материалисты, признают, что восприятие или постижение объектов является активным процессом. Объекты определяются с учетом цели мыслящего субъекта, хотя для материалиста они соответствуют заранее сложившимся в мире формам энергии, включая, несомненно, формы энергии мозга. Стул — это один из логически последовательных объектов, предназначенный для сидения (или, возможно, для скандала в баре), но не для целей субатомной физики. Теперь можно поинтересоваться, какое отношение все это имеет к объектно-ориентированному анализу. Как этот анализ связан с идентификацией объектов? Ответ состоит в концентрации внимания на пользователе.

Анализ, в центре которого стоит пользователь, при выделении объектов требует определения цели. Для возможности повторного использования необходима общая цель, поскольку объекты, разработанные для одного пользователя, могут не соответствовать требованиям другого. Фактически повторное использование является единственно возможным потому, что общество и производство определяют общий базис для восприятия. Ясное понимание онтологии как науки помогает избежать случайного, а не существенного представления объекта. Таким образом, Фред Брукс (Fred Brooks), по мнению автора, был, в некоторой степени, сведущ в онтологии или обладал этими знаниями интуитивно.

Некоторые полезные подсказки для идентификации важных, а не случайных объектов могут быть собраны в результате изучения философии, особенно гегелевской философии и современной феноменологии. В работе [728] очень подробно анализируется гегелевская концепция объекта. Основное различие между этим представлением об объектах и другими понятиями заключается в том, что объекты не являются ни произвольными "связками" свойств (эмпирическое или кантианское представление), ни базирующимися на таинственной сущности. Это концептуальные структуры, представляющие универсальные абстракции. Практическое значение этого представления состоит в том, что оно позволяет проводить различия между подлинными абстракциями высокого уровня, такими как человек, и полностью случайными, как красные объекты. Объекты могут оцениваться в соответствии с различными, исторически детерминированными категориями. Например, утверждение "эта роза красная" является качественным. Существенные для объектно-ориентированного анализа суждения представлены в табл. 6.2.

Таблица 6.2. Анализ суждений

Тип суждения	Пример	Характеристика
Качество	этот шар красный	атрибут
Рефлексия	эта трава является лекарственной	отношение
Категория	Фрэд — это человек	обобщение
Значение	Фрэд должен быть добрым	правила

Категориальное утверждение отражает абстракции высокого уровня. Такие абстракции называются **существенными** (essential). Качественные утверждения только выявляют возможные и случайные свойства, которые вряд ли подлежат повторному использованию, но, тем не менее, семантически важны в пределах данного приложения. Необходимо остерегаться таких абстракций, как "красные розы" или "опасные игрушки"; они являются качественными абстракциями и, вероятно, не могут быть повторно использованы без внутренней реструктуризации. Объекты, раскрываемые в соответствии с качественными утверждениями, называют **случайными** (accidental). Случайные объекты — это простые связки произвольных свойств, таких как "дорогие, колючие, красные розы, завернутые в фольгу". Существенные объекты универсальны в том смысле, что они соответствуют объектам, которые уже были идентифицированы социальной практикой и устойчивы во времени и пространстве. Чем они являются, зависит от социальных целей. Рефлективные суждения полезны для установления отношений использования и методов. Отношение "являться лекарственными" связывает травы с болезнями, которые они лечат. Суждения-значения могут выходить за рамки компьютерной системы, но выявлять семантические правила. Например, на очень высоком уровне анализа бизнес-процессов может быть сформулировано следующее суждение: "служащие должны быть вознаграждены за преданность", которое на более низком уровне приводится к правилу: "если человек работает не менее пяти лет, то ему к ежегодному отпуску добавляется три дополнительных дня".

Атрибуты — это функции, значениями которых являются объекты, а их диапазонами — классы. Можно выделить атрибуты с абстрактными (или существенными в вышеупомянутом смысле объектами) и случайными значениями. В первом случае это объекты, аналогичные служащему, а во втором — "красные" объекты. Это наблюдение было также сделано в контексте семантических моделей данных в [399].

Для проектирования, в центре которого находятся бизнес-процессы и пользователь, онтологическое представление диктует необходимость формулировки цели для объектов. Операции также должны иметь цель. В некоторых методах это достигается путем определения постусловий. Эти условия должны быть установлены для каждого метода (как в языке Eiffel) и для объекта в целом в разделе правил.

В [483] высказывается мнение, что экземпляры являются предметами, о которых может быть сказано нечто определенное, и обращается внимание на опасность слишком больших надежд на структуру естественного языка. Авторы этой работы считают, что понятие носит абстрактный характер и должно быть описано с помощью класса в том случае, если

- в целом об этом понятии может быть сказано несколько интересных вещей;
- имеются свойства, не используемые совместно с каким-либо другим классом;

- имеются операторы, которые отличают этот класс от некоторого большего класса, которому он принадлежит;
- границы понятия неточны;
- количество "элементов одного уровня" (например, дополнительных классов, объединение которых является их естественным обобщением) незначительно.

Они также подчеркивают точку зрения автора, что цель является главным определяющим фактором класса или типа.

Получение ответа на вопрос, *можно ли сказать об объекте больше или достаточно перечислить его атрибуты и методы*, является полезным практическим способом распознавания существенных объектов. Простое добавление большего количества свойств является мошенническим приемом в использовании этого способа. Примеров объектов такого типа имеется в изобилии. В системе по расчету заработной платы служащий может иметь рыжие волосы, хотя это не является атрибутом, или уметь управлять самолетом, хотя это не является методом. Ничего определенного не может быть сказано о таком классе: "служащие, которые могут летать", если, конечно, речь не идет о платежной ведомости для авиалинии. Ответ на вопрос, *что существенно*, является контекстно-зависимым.

Очень объемные методы, объекты с сотнями атрибутов и/или сотнями методов указывают на попытку смоделировать нечто непостижимое для простого смертного. Это подсказывает автору и, как он надеется, менеджеру проекта тоже, что аналитик не выслушал пользователей.

Объекты, предлагаемые аналитиками, затрагивают цели не только непосредственных пользователей, но и всего сообщества пользователей и, несомненно, разработчиков программного обеспечения, которые будут многократно использовать эти объекты. Поэтому аналитики должны помнить о возможности повторного использования в течение всего процесса определения технических требований. Проектирование или анализ не является копированием знаний пользователя и эксперта. Как и в случае восприятия, это является творческим действием. Проектировщик, аналитик или специалист по технике представления знаний берет цели и представления (знания) пользователей и преобразовывает их. Он или она не являются *tabula rasa*, или чистым листом, на который записывается знание, как обычно рекомендовалось старыми учебниками по определению знаний, это творческий участник.

В [423] предлагается несколько советов относительно того, в каких случаях необходимо создавать новый класс, а не добавлять метод к существующему классу, что, кажется, соответствует онтологическим представлениям этого раздела.

Эпистемология изучалась специалистами по технике представления знаний, занимающимися построением экспертных систем. Многие из выученных уроков и методик, которые они для себя открыли, могут быть использованы для построения традиционных систем. В частности они могут быть применены к проектированию человеко-машинного интерфейса [422] и в рамках объектно-ориентированного анализа и проектирования.

6.4.2. Анализ задачи

Несколько разработанных специалистами по технике представления знаний методов, предназначенных для получения знаний из людей с целью построения экспертных систем, могут использоваться для получения концептуального представления о любой области. Эти понятия часто становятся объектами. Данная книга — не место для толкования методов приобретения знания, но необходимо упомянуть полезность методов, основанных на решетках

Келли (Kelly grid) (или устойчивых решетках), анализе протокола, анализе задачи и теории интервьюирования. Использование методики решеток Келли для идентификации объектов объясняется в этом разделе ниже. Анализ протокола [262] до некоторой степени подобен обрисованной выше процедуре анализа частей речи, а анализ задачи может выявить как объекты, так и их методы. Анализ задачи часто используется при проектировании пользовательского интерфейса [213, 422]. Теория интервьюирования будет рассматриваться в главе 8.

В общих чертах, анализ задачи — это функциональный подход к извлечению знаний, включающий разделение проблемы на иерархию задач, которые должны быть выполнены. В общем случае цели анализа задачи могут быть представлены как определение

- целей задачи;
- используемых процедур;
- любых действий и вовлеченных в них объектов;
- времени выполнения задачи;
- частоты операций;
- возникновения ошибок;
- содержания задач более низкого и более высокого уровней.

Результатом является описание задачи, которое некоторым способом может быть формализовано, например, блок-схемами, логическими деревьями или даже формальной грамматикой. Однако этот процесс не описывает непосредственно знание. Иными словами, он не пытается фиксировать основную структуру знания, но пробует представить, каким образом выполняется задача и что необходимо для достижения ее цели. Только случайно выявляются какие-либо концептуальные или процедурные знания, а также какие-либо объекты.

Обычно до перехода к последующей стадии анализа протокола во время анализа задачи разрабатываются ограничивающие условия для целей по решаемой проблеме. Метод состоит в классификации факторов и идентификации элементарных "задач", связанных с решаемой проблемой. Категории, применяемые к отдельной задаче, могут быть следующими.

- занимаемое время
- частота выполнения
- используемые процедуры
- используемые действия
- используемые объекты
- частота ошибок
- положение в иерархии задач

Это также заключает в себе необходимость идентификации действий и типов таксономическим способом. Например, изучение игры в покер можно начать с определения следующей приблизительной структуры.

```
Types: Card, Deck, Hand, Suit, Player, Table, Coin
Actions: Deal, Turn, See, Collect
```

В одном из видов анализа задачи предполагается, что понятия получаются в результате объединения действий с типами, например "See player" (видеть игрока), "Deal card" (сдавать карты). После идентификации этих понятий необходимо определить планы или цели (выиграть игру, сделать деньги) и стратегии (блефовать наугад) и использовать этот анализ для идентификации необходимого знания, применяемого соответствующими парами действие-объект для описания задач. Выше упоминалось о важности идентификации объектов относительно целей.

В качестве одного из средств разбиения предметной области на составляющие подзадачи анализ задачи используется наряду с анализом потоков данных или моделированием сущностей. Хотя этот метод действительно включает анализ объектов, связанных с каждой задачей, ему недостает графических способов представления этих объектов и поэтому он остается главным образом полезным для выявления функций.

Рекомендованный в [113] подход к когнитивному анализу задачи, базирующийся на теории обработки информации человеком, менее функционален, чем обрисованный выше основной подход к анализу задачи, сконцентрированный на анализе концепций. На второй стадии трехшаговой стратегии необходимо сначала определить отношения между понятиями с помощью анализируемых примеров, а затем достроить исходную схему, анализируя укрупненные группы проблем. Схема, полученная в результате этого анализа, является моделью структуры знаний эксперта, подобной той, которая достигается в соответствии с методами концептуальной сортировки. Методы концептуальной сортировки связаны с решетками Келли, описывающими формирование экспертом фрагментов знания. Формирование фрагментов выполняется на основе идеи вероятности в соответствии с теорией обработки информации человеком, т.е. выбором корректных стимулов для решения проблемы и знанием, как оперировать этими стимулами. Как показано в [741], этот подход является родственным представлениям, касающимся моделирования объектов, благодаря сосредоточению внимания на анализе понятий и отношений до перехода к дальнейшему анализу функций/задач.

Задача — это конкретный экземпляр процедуры, которая достигает цели. Может быть множество задач, которые достигают одной и той же цели. Прецеденты — это примеры задач; они должны всегда иметь цель. В конечном счете, из выявленных задач можно получить объектную модель бизнес-процессов.

Метод иерархического анализа задачи может быть лучше понят на примере. На рис. 6.44 показано разработанное автором проектное решение. Проект касается выбора представления входных данных для финансового торговца. Полная задача, которую следует описать, — это регистрация сделки, когда торговец (в данном случае — торговец акциями) соглашается купить у некоторой стороны или продать ей несколько акций.

Диаграмма на рис. 6.44 должна интерпретироваться начиная с верхнего уровня, который представляет полную задачу заключения сделки и ее регистрации, и далее вниз слева направо. Следует обратить внимание на то, что иерархия в некоторых пунктах представлена в виде сети. Это сделано только для того, чтобы избежать повторения общих операций более низкого уровня и рекурсии.

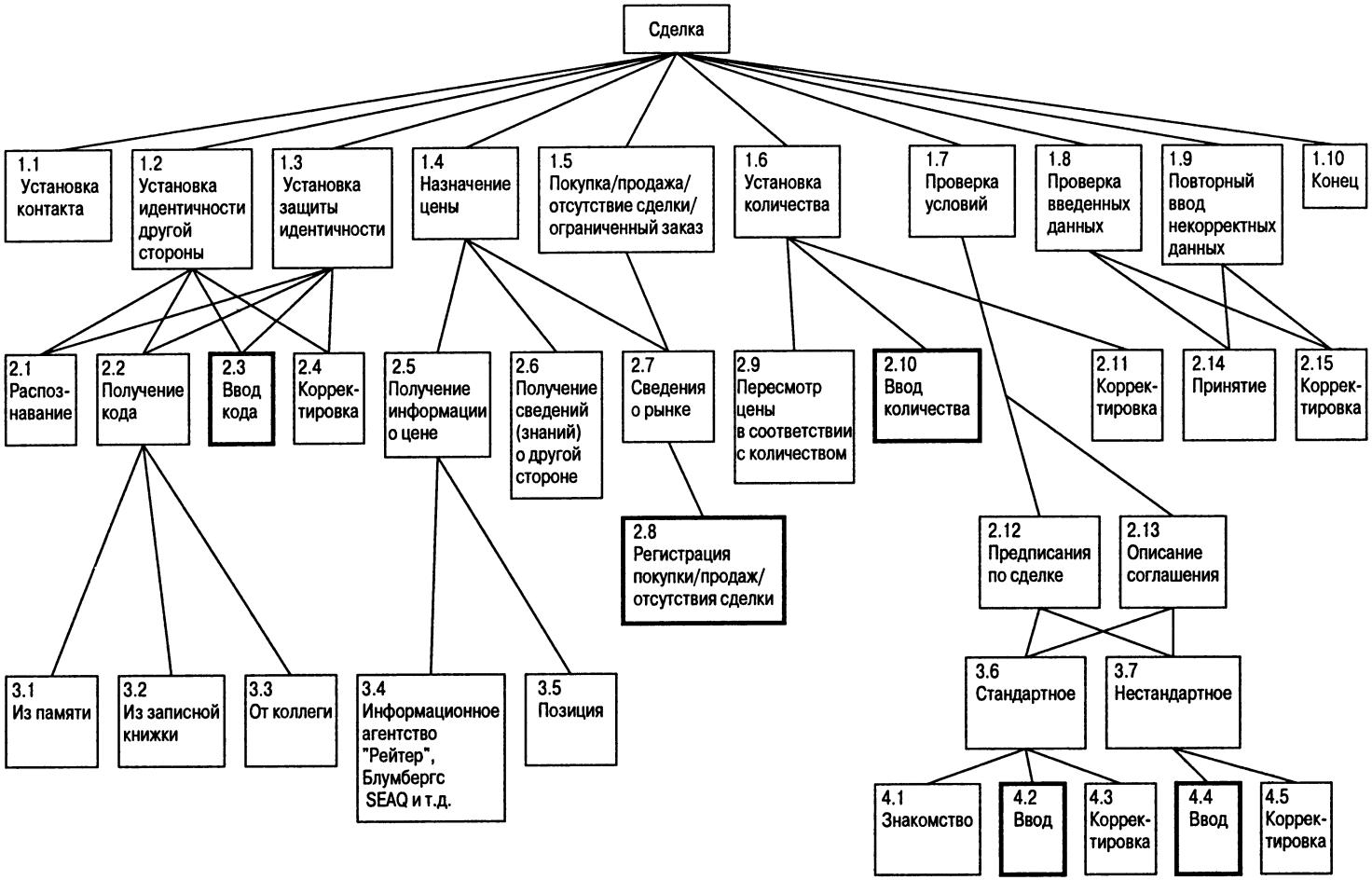


Рис. 6.44. Декомпозиция задачи заключения сделки торговцем акциями. Обратите внимание на то, что и объекты, и действия идентифицированы

Задача верхнего уровня разделена на десять шагов, начиная с начального контакта. Это, конечно, относится к инициированию (обычно по телефону) общения, при котором определяются обстоятельства сделки. Как только инициализирована эта сложная деятельность, следующий шаг (возможно, выполняемый параллельно) — это установить, с кем говорит дилер, и именно здесь на передний план выступает компьютерная система. Таким образом, эта задача требует дальнейшей декомпозиции. Например, п. 2.1 может касаться распознавания чьего-то голоса или определения его идентичности. Затем должна быть подтверждена идентичность другой стороны. Данные для идентификации (кодовый номер) может вспомнить дилер или они могут быть найдены на бумажных листах со списками, сохраняемых под рукой, или получены от коллеги, который, в свою очередь, может воспользоваться этими же источниками. Таким образом, имеются рекурсивные процессы, неявные в диаграммах, поскольку они не развернуты. Если код не получен, то процесс может закончиться (1.10) или может быть создан новый код. По получении кода он должен быть введен в систему ведения учета, компьютеризированную или сопровождаемую вручную. Это первый пункт, где есть существенное взаимодействие с технологией сбора данных сделки, а жирно вычерченный прямоугольник определяет интерфейс этого взаимодействия. Дальнейшая декомпозиция в терминах, *в частности,* движений пальца, руки и глаз не показана на диаграмме, поскольку она будет зависеть от выбранной технологии. Теперь код должен быть подтвержден, и, если он некорректен, происходит возврат к п. 2.1 или корректируется значение (п. 2.4) до возвращения к п. 2.3. Из этого пункта можно вернуться на уровень 1 и перейти к аналогичному процессу идентификации продаваемых средств. Обратите внимание на то, что этот процесс возвращений в п. 2.1 через п. 2.4 объясняет изобилие нисходящих линий в диаграмме. Подобным способом может быть интерпретирована оставшаяся часть диаграммы.

Кстати, можно заметить, что п. 2.8 следует непосредственно из п. 1.5 и что п. 1.8 относится к проверке всей сделки.

Можно заметить, что большинство задач включает сложные когнитивные процессы, в особенности такие, как пп. 1.1, 1.4 и 1.5. Однако преимуществом анализа задачи является то, что он в своей основе допускает абстракции тех задач, где есть связь с технологией сбора данных для сделки.

Этот довольно подробный пример включен сюда, чтобы дать почувствовать практические проблемы, возникающие при методике декомпозиции задачи. В частности, должно быть ясно, что использование анализа задачи важно при проектировании систем и интерфейсов пользователя, но автор утверждает, что и другие приложения, в которых необходимо концептуализировать процедурное знание, могут извлечь выгоду из этого подхода. В этом примере также показан способ, с помощью которого анализ задачи позволяет обнаружить, до некоторой степени случайно, наличие таких объектов, как, например, информация о цене и регулирующие предписания. Таким образом, для приложений, в которых функции более очевидны для понимания, чем объекты и концепции, анализ задачи является полезным способом ведения объектно-ориентированного анализа. Это часто справедливо для задач, оперирующих большим количеством неартикулированного, скрытого или скомпилированного знания. В тех случаях, где это полезно, сценарии задачи могут быть преобразованы в древовидные структуры анализа задачи.

Анализ задачи не может помочь при объединении многих психологических факторов, которые всегда имеются при сборе данных для сделки или в подобных процессах и которые являются часто совершенно неизмеримыми.

Читатели, знакомые с системой разработки Джексона (Jackson System Development), могут обратить внимание на сходство между декомпозицией задачи, представленной на

рис. 6.44, и JSD-диаграммой структуры сущности. Она, возможно, была получена в результате рассмотрения упорядоченных по времени действий жизненного цикла "сделки". Любое общее поддерево может быть "расчленено" (использован термин JSD) и рассмотрено в качестве отдельной группы действий или соответствующего ему нового объекта. Это может затем стать результатом "JSD-операции" или, другими словами, методом. Автор благодарен Дэвиду Ли (David Lee) из корпорации GEC за то, что он указал ему на это сходство. Это наблюдение обнаруживает удивительный факт, что различные дисциплины в рамках информационных технологий продолжают повторно изобретать то же самое колесо под различными названиями.

В определенной мере это подтверждает то, что использование формальной методики, такой как анализ задачи в приведенном выше примере, не может добавить ничего такого, что не может быть получено на основе здравого смысла. Однако его использование в структурировании информации, полученной из интервью, неоценимо по следующим причинам. Как видно из анализа, приведенного выше, во-первых, декомпозиция сложных задач на большее количество примитивных или унитарных действий дает возможность достигнуть лучшего понимания интерфейса между задачами и доступной технологией реализации. Это ведет к лучшему пониманию возможностей для эмпирического измерения качества интерфейса. Во-вторых, сам процесс построения и критическая оценка диаграмм иерархии задач помогают раскрывать пробелы в анализе и таким образом устраняют любые противоречия.

Анализ задачи, прежде всего, полезен при идентификации методов, а не при нахождении объектов, хотя объекты выявляются в качестве побочного эффекта. Теперь можно перейти к методам, заимствованным из инженерии знаний (knowledge engineering), которые более непосредственно касаются проблемы идентификации объектов.

В [57, 58] в контексте приобретения знаний для экспертных систем предлагается метод, который может оказаться важным для идентификации объектов, их атрибутов и методов. В этих работах приведен пример инженера по представлению знаний, который ищет эмпирические правила высокого уровня, базирующиеся на опыте (эвристике). Предположим, в области озеленения обнаруживается, что результатом регулярной стрижки травы являются хорошие лужайки. Инженер по представлению знаний не должен удовлетворяться этим, поскольку это не показывает границ компетентности заданной системы. Система, которая дает уверенный совет в областях, где она некомпетентна, не нужна. Необходимо более глубокое понимание проблемной области. Таким образом, следующий вопрос, заданный эксперту, может иметь такой вид: "почему?". Ответ может быть следующим: "Потому что регулярная стрижка уменьшает количество грубых трав и поддерживает упругость дерна". Отсюда получено два атрибута объекта "хороший дерн", чьим родителем в иерархии является, конечно, "дерн". Почему регулярная стрижка приводит к упругому дерну? Понятно, это способствует прорастанию листьев. Поскольку известны причинно-следственные знания (causal knowledge), теперь можно приступить к выявлению методов. Для помощи в определении границ Басден предлагает задавать вопросы "что еще" и "что в отношении... ". В приведенном примере инженер по представлению знаний должен спросить: "что можно сказать в отношении условий засухи?" или "что еще позволяет получить хорошие лужайки?". Эти технические приемы опроса очень полезны для аналитиков, использующих объектно-ориентированный подход.

6.4.3. Решетки Келли

Одной из наиболее полезных методик инженерии знаний для выявления объектов и их структур является методика, основанная на решетках Келли или устойчивых решетках (repertory grid). Эти решетки первоначально были введены в клинической психиатрии [438].

Их используют в качестве методики, предназначенной для помощи аналитикам в выявлении "личностных конструкций"; концепций, которые люди используют для построения своего мира и взаимодействия с ним. Конструкции — это пары противоположностей, например медленно-быстро, которые обычно соответствуют или классам или значениям атрибутов в объектно-ориентированном анализе. Второе измерение решетки — это ее "элементы", которые соответствуют объектам. Элементы оцениваются по пятибалльной шкале, согласно которой они наиболее близко соответствуют полюсу конструкции. Затем эти значения могут использоваться для "фокусирования" решетки. Фокусирование — это математическая процедура, которая раскрывает связи между элементами и конструкциями. В частности, фокусирование располагает элементы по рангу в порядке ясности, с которой они восприняты, а конструкции — в порядке их важности в качестве классификаторов элементов. Более подробная информация может быть найдена в любой приличной книге по приобретению знаний, например [331, 364].

Для того чтобы проиллюстрировать полезность решеток Келли, можно предположить, что необходимо взять интервью у пользователя. Методика включает вначале идентификацию некоторых "элементов" в приложении. Они могут быть реальными объектами или концепциями и должны быть организованы в логически последовательные множества. Например, множество {Порш, Ягуар, Ролс Ройс, Мини, Шофер} имеет очевидный лишний элемент: Шофер.

Использование методики, основанной на решетке Келли, в ее полной форме не рекомендуется. Однако методики опроса, основанные на решетках Келли, чрезвычайно мощны при выявлении новых классов и атрибутов, а также при расширении и усовершенствовании классификационных структур. Имеются три основные методики.

- вопросы о противоположностях всех элементов и концепций
- ступенчатая (laddering) методика для выделения обобщений
- выявление триад (elicitation by triads) для выделения специализации

Рассматривая рис. 6.45, можно обнаружить, что ключевым является класс `Спортивные автомобили`. Противопоставить ему можно класс семейных автомобилей (а не неспортивных), что является не логической противоположностью, а совершенно новым классом. Таким образом, спрашивая о противоположности класса, можно выявить новые классы.

При ступенчатой методике пользователей просят дать названия для понятий более высокого уровня: "Вы можете придумать слово, которое описывает все эти понятия {скорость, роскошь, экономичность}?". В данном случае результатом может быть понятие "денежной стоимости". Эта методика известна как ступенчатая, и она выявляет как классификационные, так и композиционные структуры. В общем случае результаты этой методики приводят к более общим понятиям. Задавая вопрос о термине, который обобщает понятия быстрый и спортивный, можно обнаружить класс автомобилей "для поддержки тонуса".

Выявление триад относится не к китайской пытке, а к методике, посредством которой пользователя просят выбрать из установленного логически последовательного множества элементов любых три и определить понятие, применимое к двум из них, но не к третьему. Например, в случае с триадой {Порш, Ягуар, Мини} в качестве важного может появиться понятие максимальной скорости. Точно так же триада {Мини, Ягуар, Трабант} может выявить атрибут `CountryOfManufacture`: или классы `BritishCar` и `GermanCar`. В качестве варианта этой методики можно попросить пользователей разделить элементы на две или больше групп и затем дать группам названия. Этот метод называется сортировкой карт (card sorting).

Все эти методики являются первоклассными способами для постижения концептуальной структуры предметной области, если они используются с осторожностью и тщательностью. Например, полный перечень всех триад может быть чрезвычайно утомителен, и легко заставить пользователей отвернуться от этих методик.

	Элементы					
Понятие	Rolls Royce	Porsche	Jaguar	Mini	Trabant	Обратное понятие
Экономичный	5	4	4	2	2	Дорогостоящий
Удобный	1	4	2	4	5	Основательный
Спортивный	5	1	3	5	5	Семейный
Дешевый	5	4	4	2	1	Дорогой
Быстрый	3	1	2	4	5	Медленный

Рис. 6.45. Решетка Келли. Баллы от 1 до 5. Понятиям слева соответствуют низкие баллы, а понятиям справа (их противоположность) — высокие. Решетка не фокусирована

Имеется несколько компьютерных систем, которые автоматизируют построение и фокусирование этих решеток, например система ETS [102] и ее промышленный потомок AQUINAS [103]. Эти системы преобразовывают решетки в наборы правил. Любопытно, что эти автоматизированные инструментальные средства отбрасывают многое из того, что объединяет анализ устойчивой решетки. Например, ясно, что результатом ступенчатой методики и сортировки являются классификационные структуры. Затем они маскируются под порождающие правила с последующей потерей информации. Автор в первом издании этой книги предсказывал, что будут развиваться инструментальные средства, непосредственно фиксирующие структурную информацию такого типа. Это действительно начало происходить, что было проиллюстрировано работами [288], но, насколько знает автор, работа еще не достигла промышленных масштабов. Тем временем эта методика должна использоваться вручную (и предпочтительно неформально) для объектно-ориентированного анализа.

Объектные шаблоны использовались в приобретении знаний для экспертных систем. Заполнение шаблонов является структурным методом для сбора семантической информации, которая имеет общую применимость. В подходе инженерии знаний подчеркивается, что классы должны соответствовать понятиям, поддерживаемым пользователями и экспертами. Классы высокого уровня представляют абстракции, которые могут быть повторно использованы в нескольких похожих областях применения. Абстракция "объект" универсальна во всех областях, а такая абстракция, как "финансовый отчет" (account), пригодна для использования в большинстве финансовых и бухгалтерских приложений. Ипотечный финансовый отчет более специализирован и поэтому может повторно использоваться только в узком наборе приложений. Основным умением для аналитиков, которые занимаются поисками преимуществ возможности повторного использования, является представление абстракции на нужном уровне. Создание прототипов и взаимодействие с пользователями и специалистами по предметной области помогают выявлению знаний об объектах.

Таким образом, онтология и эпистемология помогают в нахождении объектов и структур. Они также могут помочь в распознавании хороших объектов в библиотеке собственными усилиями. Третьим необходимым инструментом для этой цели является, конечно же, здравый

смысл. При этом для идентификации объекта достойны рассмотрения несколько следующих рекомендаций.

- Всегда следует помнить, что хороший объект с возможностью повторного использования представляет нечто универсальное и реальное. Объект — это социальное существо; его методы могут использоваться другими классами. В противном случае необходимо спросить, в чем заключается его функция, или удалить его из модели.
- Хотя объект не должен быть настолько сложным, чтобы представлять трудность для понимания, все же он должен инкапсулировать (скрывать) в каких-то разумных пределах сложное поведение, чтобы оправдать свое существование.
- Метод, который не использует атрибуты, принадлежащие его текущему классу, вероятно, инкапсулирован в чужом объекте, так как он не нуждается в доступе к этой частной информации.

Измерение качества абстракции — очень сложный процесс. Здесь можно провести аналогию с проектированием устройств. Как и в случае с устройствами, должно быть минимальное количество взаимозаменяемых частей, а части должны быть, насколько возможно, общими. Предложенные критерии, с их соответствующей метрикой, включают несколько критериев, которые уже встречались в контексте объектно-ориентированного программирования.

- Интерфейсы должны быть как небольшими, так и, насколько возможно, простыми и устойчивыми.
- Объект должен быть самодостаточным и законченным в соответствии с лозунгом: "Объект, весь объект и ничего кроме объекта". В качестве контрпримера этого понятия законченных объектов можно рассмотреть класс объектов, имена которых начинаются с букв "СН". Другими словами, необходимо избегать случайных объектов. Объекты не должны посылать множество сообщений для того, чтобы сделать простые вещи: топология структуры использования должна быть проста.

Подобные рекомендации применимы и к методам. Методы также должны быть простыми и общими. Например, метод "прибавить 1" порождает метод "прибавить *n*" для всех *n*. Необходимо унифицировать операции такого вида. Методы должны быть релевантными, т.е. они не должны быть применимыми именно к этому понятию, а не более конкретному или более общему. Методы должны зависеть от инкапсулированного состояния объекта, в котором они содержатся, как было упомянуто выше. Очень важным принципом объектной ориентации является принцип слабого связывания или "Закона Деметры"[9], который утверждает, что "методы класса не должны каким-либо образом зависеть от структуры какого-либо класса, кроме непосредственной (верхний уровень) структуры собственного класса. Более того, каждый метод должен передавать сообщения объектам, принадлежащим только очень ограниченному набору классов" [681]. Это помогает автономному пониманию классов, что обеспечивает возможность их повторного использования.

Следует вспомнить, что аналитики должны избегать объектов, возникающих единственно из нормализации или удаления отношений "многие ко многим". Правило заключается в следующем: если нечто не является реальной сущностью — это не объект. Например, отношения

[9] Это греческая богиня сельского хозяйства, поэтому она имеет отношение к циклическому возрождению.

"многие ко многим" между заказами и счетами могут быть удалены после введения нового класса строки заказа. Это прекрасно, строки — реальные объекты; они напечатаны на счете. Напротив, нет такого естественного объекта, который удалил бы отношение "многие ко многим" между автомобилями и цветами, в которые они могут быть окрашены.

Последнее, что следует отметить, — аналитики не должны надеяться на то, что они все правильно поймут с первого раза. Они никогда не поймут. Это ошибка каскадной модели (разрабатываемого проекта), и слишком хорошо известно, что затраты на поддержание неправильно специфицированных систем высоки. Создание прототипа и проектирование на основе задач (task-centred design) при должном управлении позволят аналитику все правильно понять, но с третьего раза.

6.5. CASE-средства

Существует несколько коммерческих CASE-средств, поддерживающих язык UML или его аспекты. Большинство из них являются графическими редакторами, которые проверяют синтаксис моделей UML и на их основе генерируют код. Возможно, самым известным из них является программа Rose от корпорации Rational Inc. Система Rose начинает работу с создания двух пакетов, один из которых предназначен для построения диаграмм прецедентов, а другой — диаграмм классов. Но это разделение условно: в любой пакет можно поместить диаграммы любого вида, объединяя классы, прецеденты и исполнителей в рамках одной диаграммы. Прецеденты можно переместить на диаграммы классов и *наоборот*. Исходя из своих предпочтений, их можно смешивать или не смешивать на одной диаграмме. Ключевые слова, инварианты и наборы правил поддерживаются не в достаточной степени, и поэтому необходимо использовать комментарии для описаний, наборов правил, инвариантов, а также пред- и постусловий.

В системе Rose диаграммы классов и состояний (statechart) — это различные диаграммы. Для одного класса допускается несколько диаграмм состояний, которые помещаются в одном документе.

Преимуществом системы Rose является удобство интеграции с другими средствами корпорации Rational Inc., предназначенными для управления конфигурацией, тестирования, документирования технических требований и т.д. Известны также инструментальные средства от сторонних производителей, такие как система RoseLink, которая дает возможность генерировать скелеты приложений на C++ или Java для ООСУБД Versant, конвертируя ассоциации в коллекции C++ или СУБД, соответствующие стандартам группы ODMG. При работе с такими CASE-средствами неизбежно возникает искушение в создании огромных диаграмм. Автор уже доказывал, что это не является хорошей идеей. Лучше написать хороший комментарий, расставив акценты с помощью небольших диаграмм, которые помогают устранять неоднозначность и иллюстрируют текст. Инструментальные средства, такие, например, как Soda, могут использоваться для помощи во встраивании диаграмм в комментарий. Помещение каких бы то ни было спецификаций прецедентов в главный прямоугольник комментария (где они более видимы) и в примыкающие примечания или в оба типа является хорошей идеей.

Наиболее впечатляющим средством этого типа является система Together, которая поставляется в вариантах для C++ и Java. Например, TogetherJ поддерживает подлинно циклическую разработку: при построении диаграммы в отдельном окне появляется код Java, а

при редактировании кода Java изменяется диаграмма. Система Rose делает кое-что подобное, но не так хорошо.

Система Paradigm Plus от компании Computer Associates и ее набор программных продуктов COOL также поддерживают создание диаграмм на языке UML. Набор средств COOL Jex поддерживает некоторые из методик Catalysis. Компонентная модель, лежащая в основе программных продуктов COOL, рассмотрена в [158].

Программа Select от корпорации Princeton Softtech, а также StP (Software through Pictures — программирование посредством рисунков) от фирмы Aionix и System Architect — это дополнительные примеры UML-совместимых CASE-средств. Имеются также средства, основанные на UML, которые поддерживают отдельные методы разработки, в том числе метод Шлеер-Меллора (Shlaer/Mellor), а не только систему обозначений UML.

Кроме поддержки языка UML и структурных методов, система System Architect обладает возможностями, которые поддерживают моделирование бизнес-процессов с помощью использования каркаса Захмана (Zachman) [725, 822]. Каркас Захмана — это широко используемая схема классификации для описательных или нотационных представлений уровня предприятия. В этой структуре обозначения размещаются в ячейках матрицы, как показано на рис. 6.46. Столбцы и строки обеспечивают законченную схему классификации (поскольку существует только шесть типов вопросов). Примечательно, что строки формируются по аналогии со строительством здания и представляют перспективы, принятые различными строительными отраслями в течение процесса строительства. Этим можно руководствоваться, но трудно понять, как это разбиение данных и функций может согласовываться с объектно-ориентированным подходом. В [327] показано, что эта структура сворачивается до меньшего количества ячеек, если понятия ООП подставляются вместо таких представлений, как "логическая модель данных".

И все же ни одно средство должным образом не поддерживает подстановку в эту структуру обозначений метода Catalysis, описанного в разделе 6.3.7, хотя Platinum Plus скоро сможет их поддерживать. Вероятно, появятся и другие средства, возможно через некоторое время похожая методика появится в стандарте языка UML. Система TogetherJ уже обеспечивает нечто довольно похожее с помощью поддержки шаблонов.

Чего недостает в сфере CASE-средств, так это хороших объектно-ориентированных средств для имитационного моделирования бизнес-процессов. Они позволили бы анимировать модели бизнес-процессов. Также ощущается недостаток в таких инструментальных средствах, которые можно было бы определить в качестве "anti-CASE-средств", способных автоматически генерировать большую часть UML-диаграмм. В идеальном случае эти средства должны быть интегрированы в единый пакет и иметь интерфейсы с популярными традиционными CASE-средствами и графическими пакетами.

Автор предпочитает использовать для построения диаграмм дешевый инструмент Visio и программу PowerPoint. По его мнению, они менее "назойливы" и позволяют вводить новшества, где это необходимо. Шаблоны языка UML легко доступны для Visio.

	Данные (Что)	*Функция* (Как)	*Сеть* (Где)	*Люди* (Кто)	*Время* (Когда)	*Мотивация* (Почему)
Область действия (Планировщик)	Список предметов	Список процессов	Список местоположений	Список организаций	Список событий	Стратегические цели и т.д.
Бизнес-модель (Владелец)	 Диаграмма "сущность-связь" Семантическая модель	Блок-схема процесса	Логистическая сеть	Органограмма	Бизнес-график	Бизнес-план
Системная модель (Проектировщик)	Логическая модель данных	 Архитектура приложения на основе диаграмм потоков данных DFD (Data Flow Diagram)	Архитектура распределенной системы	Архитектура человеко-машинного интерфейса	Структура обработки	Архитектура знаний
Технологическая модель (Разработчик программы)	Проектирование данных	Проектирование системы (Структурная диаграмма)	Системная архитектура	Проектирование человеко-машинного интерфейса	Управляющая структура	Проектирование знаний
Детальное представление (Субподрядчик)	Определение данных	Программа	Сетевая архитектура	Безопасность, архитектура	Прерывания и т.д.	База знаний

Рис. 6.46. Каркас Захмана с примерами представлений

6.6. Резюме

В этой главе представлена краткая история объектно-ориентированного анализа и объектно-ориентированных методов проектирования, проводятся различия между трансляционным подходом и подходом детальной разработки, а также между методами, основанными на данных и на обязанностях. Автор привел аргументы, что лучшими являются трансляционные методы, основанные на обязанностях.

Основы объектно-ориентированного анализа и объектно-ориентированного проектирования были представлены с использованием системы обозначений языка UML, и в качестве наилучшей практики был аргументированно рекомендован метод Catalysis. Были рассмотрены его концепции уточнения, отражения, извлечения и каркасов. Значительный акцент был сделан на роли инвариантов и наборов правил, введенных в методе SOMA.

Также подробно рассматривались методы идентификации объектов, включая несколько методов, берущих начало в выявлении знаний для экспертных систем.

В конце был представлен краткий обзор современных CASE-средств.

6.7. Дополнительная литература

Трактовка объектно-ориентированного анализа и проектирования в этой главе базировалась главным образом на методах Catalysis и SOMA, а также на языке UML. Это понимание выработано в результате изучения результатов многих специалистов по методологии разработки. Большая часть этих исследований рассмотрена в приложении Б, где приводятся ссылки на первоначальные источники.

В работе [204] метод Catalysis определен как метод объектно-ориентированной и компонентно-ориентированной разработки, что имеет смысл благодаря языку UML.

Метод SOMA (The Semantic Object Modelling Approach), как подход к семантическому моделированию объектов, возник из попыток комбинирования объектно-ориентированного анализа с идеями моделирования бизнес-процессов и систем, основанных на использовании знаний [312, 327, 329]. Эта книга переопределяет SOMA как расширение метода Catalysis за счет введения наборов правил и добавления технологии разработки технических требований, представленной в главе 8, а также инноваций процессов в главе 9. Большая часть материала раздела 6.3.4 основана на [334].

Хороший, краткий и популярный обзор языка UML содержится в [282]. На эту книгу, как на первоисточник, ссылаются в работах [101] и [675]. Наиболее современная справочная информация может быть найдена в Internet по адресу: `www.omg.org`.

В [485] рассматриваются различные интерпретации наследования в инженерии знаний и объектно-ориентированном программировании. В этой работе изложен интересный взгляд на проблему множественного наследования.

6.8. Упражнения

1. Сколько методов или фрагментов методов объектно-ориентированного анализа и проектирования было опубликовано?

   а) 12
   б) между 13 и 25
   в) 17
   г) между 26 и 44
   д) 44
   е) более 44

2. Какой из следующих методов объектно-ориентированного анализа и проектирования связан с именем Джима Румбаха (Jim Rumbaugh)?

   а) CRC
   б) HOOD 25
   в) Ptech
   г) Objectory
   д) OMT

3. Определите "объект" (одним предложением). Назовите четыре составляющие объекта.
4. Каковы отличия между типами, классами, экземплярами и ролями?
5. Что такое аспект (facet)? Приведите три совершенно разных примера.
6. Почему анализ наиболее важен для объектно-ориентированных систем?
7. Определите анализ, логическое проектное решение и физическое проектное решение. Каково различие между анализом и логическим проектным решением, между логическим и физическим проектным решением?
8. Назовите десять объектно-ориентированных методов или систем обозначений (*подсказка*: может быть полезно приложение Б).
9. Перерисуйте структуру наследования, представленную на рис. 6.16, уделив должное внимание различным дискриминаторам.
10. Каково различие между оболочкой и подсистемой или пакетом в большинстве методов объектно-ориентированного анализа?
11. Напишите инварианты, которые могут быть применены для всех циклов, представленных на рис. 6.19.
12. Назовите три объекта, для которых может использоваться набор правил.
13. Назовите шесть типов объектов, которые могут использоваться повторно.
14. Что такое предусловие, постусловие, утверждение, условие инвариантности, инвариант класса, гарантия, надежность?
15. Сравните три известных объектно-ориентированных метода.
16. Как выделить атрибут или класс?
17. Двунаправленные ассоциации нарушают инкапсуляцию. Почему? Каким образом может быть преодолена эта проблема? Верно ли это и при анализе и при проектировании? Если нет, то почему?
18. Рассмотрите использование нормализации в объектно-ориентированном и обычном моделировании.
19. Рассмотрите применение функциональной декомпозиции в объектно-ориентированном и обычном моделировании.
20. Запишите небольшой набор правил, определяющий поведение технического аналитика (специалиста по прогнозированию биржевой конъюнктуры), имеющего дело только с безопасностью.
21. Когда и для каких целей могут использоваться модели состояний?
22. Приведите пример вопроса, который позволяет отличить абстрактный класс от двух более конкретных классов. Приведите пример вопроса, который позволяет выявить наиболее конкретные классы из списка объектов. Приведите пример вопроса, который поможет выявить еще неупомянутое понятие.
23. Приведите некоторые простые рекомендации для анализа текстов.
24. *Существование типа не зависит от существования его экземпляров.* Обсудите это высказывание с учетом взглядов Платона.
25. Почему анализ важен для объектно-ориентированного программирования?

26. Генерирует ли Бог исключительные ситуации?
27. С использованием описанного в этой главе подхода определите одну из систем.

    а) простую систему администрирования общедоступной библиотеки, книги которой можно получать и возвращать

    б) простую графическую программу, позволяющую рисовать, перемещать, удалять и группировать изображения

7

Архитектура, шаблоны и компоненты

вместе с Аланом О'Каллаганом и Аланом Камероном Уилсом

> *Действия паука похожи на работу ткача, а пчела, конструирующая соты, делает это лучше многих архитекторов. Но что отличает самого плохого архитектора от наилучшей из пчел — это то, что архитектор перед построением структуры в реальной жизни представляет ее в своем воображении.*
>
> К. Маркс. *Капитал*, том I

В этой главе исследуются различные проблемы, с которыми сталкиваются специалисты в процессе современной разработки объектно-ориентированных систем. В частности, здесь речь пойдет о появлении такой дисциплины, как архитектура программного обеспечения, и будут кратко описаны продолжающиеся дебаты проектировщиков объектно-ориентированных систем, которые, вероятно, открыли для себя больше возможностей, чем подавляющее большинство разработчиков программного обеспечения. Далее в контексте этой дискуссии исследуем шаблоны, которые считаются необходимыми для хорошего объектно-ориентированного решения. И наконец, рассмотрим применение объектно-ориентированных идей в разработках, основанных на компонентах. В этой области можно реализовать почти все те преимущества, о которых шла речь в главе 2. Как и в главе 6, наша трактовка разработки, основанной на компонентах, во многом совпадает с методом Catalysis.

7.1. Архитектура программного обеспечения и систем

Архитектура программного обеспечения является ключевой составляющей как разработки программного обеспечения вообще, так и объектно-ориентированных технологий в частности. Начиная с 1992 года ее значимость ощутимо возрастает (в силу причин, которые мы рассмотрим ниже) благодаря четкому разделению на различные школы мышления, внутри которых и произошло позднее ее естественное появление. Однако модельное представление имеет более глубокую генеалогию по сравнению с той, которую порой готовы признать современные специалисты. Понятие архитектуры уходит корнями к таким понятиям, как "разработка программного обеспечения" и "кризис программного обеспечения". Появление этих двух терминов обычно связывают с известной конференцией NATO, прошедшей в 1968 году [582]. Фред Брукс (Fred Brooks) упоминает архитектуру в своей известной работе о мифических человеко-месяцах [118]. Однако он связывает первое применение этого термина с работой [80], написанной пятью годами ранее. Однако согласно утверждениям Яна Хьюго (Ian Hugo), присутствовавшего в качестве делегата на этой исторической конференции NATO, идея "архитектуры программного обеспечения" и роль "архитекторов программного обеспечения" были широко озвучены в дискуссиях; однако аналогия казалась редакторам докладов слишком странной, чтобы подобное понятие могло быть отражено в официальных докладах. В самом деле символично, что данная идея имеет такое же значение, как и идея кризиса программного обеспечения, и что она почти не упоминалась в последующий период, тогда как дисциплина программной инженерии, применявшаяся на практике в течение более трех десятилетий, перетерпела неудачу при разрешении поставленных перед ней проблем. Вероятно, не вызывает удивления тот факт, что на современном этапе архитектура программного обеспечения находится в центре дебатов, т.е. переоценивается сама суть дисциплин разработки программного обеспечения и систем. И в ходе подобных дебатов опыт применения как объектно-ориентированных технологий, так и шаблонов программного обеспечения играет чрезвычайно важную роль.

Архитектура как крупная структура

До настоящего времени все еще нет четкого и точного определения, что же такое архитектура программного обеспечения. То соглашение, которое действительно существует на сегодняшний день, как оказалось, относится к проектному решению высокого уровня и крупным системам, включающим как статические, так и динамические свойства. Например в [59] приводится следующее определение (с тех пор часто цитируемое).

> Архитектура программы или вычислительной системы — это структура системы, которая состоит из программных компонентов, внешних видимых свойств этих компонентов и отношений, существующих между ними.

Мэри Шоу (Mary Show) и Дэвид Гарлан (David Garlan) в своей известной работе, посвященной архитектуре программного обеспечения, заходят настолько далеко, что связывают ее появление с разбиением программных систем на модули [704]. В то же время авторы признают, что исторически такие архитектуры явно не выделялись. Пытаясь представить архитектуру более явно и формально, они вводят важные понятия *архитектурного стиля* и *языка описания архитектуры ADL* (Architectural Description Language).

Они трактуют архитектуру конкретной системы как набор вычислительных компонентов с описанием их взаимодействия (или соединения). Согласно Шоу и Гарлану, архитектурный стиль определяет семейство систем на основании их структурной организации. Клиенты, серверы, фильтры, уровни и базы данных — все это возможные примеры компонентов, тогда как примерами соединений могут служить обращения к процедурам, передача событий, протоколы баз данных и каналы. В табл. 7.1 (представляющей собой адаптированную версию таблицы, содержащейся в книге [704]) приводится список распространенных архитектурных стилей.

Позицию Шоу и Гарлана, а также мнение других специалистов из Университета Карнеги Меллона и Института программной инженерии SEI (Software Engineering Institute), в котором работают авторы [59], можно охарактеризовать как представление архитектуры в виде структуры системы высокого уровня. Включение в таблицу объектно-ориентированных систем уже само по себе выводит на первый план вопрос, связанный с подходом, который сводит архитектуру к одному из *компонентов с соединениями*. Многие полагают, что архитектура представляет собой нечто большее, чем просто прямоугольники и линии, с которыми, однако, связывается значительная часть семантики. Чтобы такая схема имела смысл, она должна включать понятие архитектурного в*и*дения: унифицированное понятие, которое все мы понимаем одинаково, когда видим элегантную программную систему или, что еще более важно, когда должны построить такую систему.

Таблица 7.1. Распространенные архитектурные стили (согласно Шоу и Гарлану)

Системы потоков данных	**Виртуальные машины**
Пакетная обработка	Интерпретаторы
Конвейеры и фильтры	Системы, основанные на правилах
Системы вызова и возврата	**Системы, основанные на данных (репозитарии)**
Основная программа и подпрограмма	Базы данных
Объектно-ориентированные системы	Гипертекстовые системы
Иерархические уровни	Методология “Классной доски”
Независимые компоненты	
Взаимодействующие процессы	
Системы событий	

Шоу и Гарлан фокусируют свое внимание на технологии реализации и рассматривают объекты просто как экземпляры абстрактных типов данных. Они отмечают, что объект отвечает не только за хранение своего представления, но и за сокрытие этого представления от своих клиентов. Тем не менее они не поняли важности этого вопроса. Некоторые авторы, например [186, 597], наряду со многими другими специалистами обращали внимание на тот факт, что объект сам по себе представляет абстрактное поведение (через свои операции), инкапсулирующее как реализацию своих операций, так и (локализованные) данные, которые он

обрабатывает, и поэтому изолирует программиста от цифровой архитектуры используемой им машины. Вследствие этой независимости от ограничений Фон Неймана (Von Neuman) разработчики архитектуры могут использовать объекты как строительные блоки, что позволяет создавать архитектуры программного обеспечения в любом количестве, и особенно такие архитектуры, в которых отражена терминология предметной области. Кук считает, что в этом смысле объекты являются "архитектурно независимыми". Помимо всего прочего, это означает, что объекты могут использоваться для создания конвейерных структур, многоуровневых архитектур, систем "классной доски" и любых других стилей, представленных в приведенной выше таблице.

Причисление объектно-ориентированного подхода просто к одному из нескольких стилей реализации позднее не было поддержано в [329]. В этой работе объекты рассматриваются как "общий механизм приобретения знаний". Это значит, что использование объектов в других областях, не относящихся к реализации, не предусматривается, и поэтому их потенциальная значимость для архитектуры программного обеспечения сильно уменьшена. В этом состоит новый взгляд на понимание архитектуры программного обеспечения, так как зачастую специалисты ограничивались только возможностью моделирования объектов и перспективными системами, которые в результате должны проявлять гибкость по отношению к изменениям бизнес-процессов, что и делало их привлекательными. Когда Шоу и Гарлан сравнивают, как они называют, архитектуру "неявного инициирования" (т.е. системы, основанные на исполнителях, или системы "классной доски", где объекты регистрируют заинтересованность в событиях) с объектной технологией, они, как оказалось, не учитывают, что неявное инициирование легко моделируется при помощи объектно-ориентированных методов. Их представление объектно-ориентированного подхода, судя по всему, ограничивается моделями современных объектно-ориентированных языков программирования. Однако все в той же работе они применяют методы объектно-ориентированного проектирования для своего средства архитектурного проектирования AESOP — неявно допуская таким образом, что объектно-ориентированная методология позволяет описывать другие стили. Еще хуже то, что они критикуют объектно-ориентированный подход за неспособность представлять стилистические ограничения. Такое узкое представление объектно-ориентированной технологии исключает возможность применения объектно-ориентированных методов, согласно которым объекты, обладающие встроенными наборами правил, могут представлять такие понятия, как семантические ограничения целостности и т.п.

С другой стороны, Шоу и Гарлан действительно предоставляют некоторые обоснованные аргументы в пользу расширения объектно-ориентированного модельного представления. Они утверждают, например, что языки описания архитектуры (ADL) предполагают необходимость моделирования ролей (а значит, и динамической классификации). Конечно, это не означает, что объекты лучше всего описывают любые требования. Исследование архитектуры многократного использования, выполненное в компании Tektronix Inc., продемонстрировало, что различные топологии характеризуются различными затратами и преимуществами и, следовательно, разной применимостью. Но авторы этого исследования несколько неубедительно утверждают, что компоненты и соединители дополняют архитектуру. Для практикующих специалистов из области объектной технологии подобное редукционистское представление архитектуры оказывает весьма незначительную помощь при создании программных систем. В распоряжении разработчиков имеется множество различных архитектурных возможностей, которыми они могут воспользоваться в любой момент.

Еще одно немаловажное замечание, сделанное в книге Шоу и Гарлана, связано с появлением языков описания архитектуры (ADL). Авторы утверждают, что структурная декомпозиция

традиционно может быть выражена либо при помощи средств создания модулей, присущих языкам программирования, либо с использованием специального языка межмодульного соединения MIL (Module Interconnection Language). Их критикуют за то, что они дают слишком низкоуровневое описание соединений между различными вычислительными элементами, а также за то, что "они не достигают успеха в представлении четкого разделения задач на уровне архитектуры, а также тех задач, которые связаны с выбором алгоритмов и структур данных". Новейшие языки, основанные на компонентах, например Occam II [640] и Connection [514], или среды, например STEP [668], которые ввели в обиход специализированные структурные шаблоны, критикуют за ограниченную область действия. В настоящее время сформулировано шесть следующих ключевых требований, предъявляемых к языку более высокого уровня — ADL (языку описания архитектуры).

1. *Композиция*. Должна существовать возможность описания системы в виде композиции, состоящей из независимых компонентов и соединителей.
2. *Абстракция*. Должна существовать возможность описания компонентов и их взаимодействия в рамках программной архитектуры таким способом, который явно и подробно обозначает абстрактные роли компонентов в системе.
3. *Возможность повторного использования*. Должна существовать возможность многократного использования компонентов, соединителей и архитектурных шаблонов в различных описаниях архитектуры, даже если они были разработаны вне контекста данной архитектурной системы.
4. *Конфигурация*. Описания архитектуры должны включать информацию о структуре системы, причем независимо от используемых в этой структуре элементов. Кроме того, такие описания должны поддерживать динамическое реконфигурирование.
5. *Гетерогенность*. Должна существовать возможность комбинировать несколько разнородных описаний архитектуры.
6. *Анализ*. Должна существовать возможность выполнения глубокого и разностороннего анализа описаний архитектуры.

Шоу и Гарлан достаточно точно отмечают, что типичные диаграммы, которые состоят из прямоугольников и линий и нередко рассматриваются как "описание архитектуры", фокусируют внимание на компонентах. Поэтому они не предусмотрены в некоторых ключевых контекстах. Это касается компонентов-оболочек от сторонних производителей, мультиязыковых систем, уже существующих систем и, что, возможно, самое важное, всех крупномасштабных встроенных систем реального времени. Кроме того, прямоугольники, линии и элементы, находящиеся между прямоугольниками, не всегда способны обеспечить семантическую непротиворечивость при переходе от одной диаграммы к другой (а иногда даже в рамках одной и той же диаграммы). Они не предполагают определения структурных интерфейсов. Это свидетельствует, как минимум, о двух уровнях структуры и абстракции (как правило, отсутствующих): абстракции соединений и сегментации интерфейсов. Подобные диаграммы в значительной степени зависят от знаний и опыта специалиста, выполняющего роль архитектора, и все это поддерживается неформальным образом.

Универсальный язык соединителей UniCon (Universal Connector Language) представляет собой пример языка описания архитектуры (ADL), разработанного в Университете Карнеги Меллона [702]. Помимо всего прочего, для него характерно наличие явно указанного структурного элемента — соединителя, представляющего правила, предназначенные для соединения

компонентов. Он обладает некоторой степенью абстракции, которая позволяет получить описание архитектуры на основании рассмотрения низкоуровневых компонентов структуры. Компонент имеет видимый и известный интерфейс; соединитель связан с ролями — именованными составляющими его протокола — которые должны исполняться интерфейсом компонента. Язык UniCon поддерживает проверку ассоциаций между интерфейсом компонента и ролями, проверку типов самих компонентов и соединителей, а также строгое соответствие компонентов и соединителей архитектурным стилям.

Язык UniCon и другие языки описания архитектуры (ADL), несомненно, способствовали созданию более строгих и формальных описаний и методов анализа структуры, что имеет первостепенное значение для некоторых классов приложений, особенно в системах реального времени. В работе [694] приводятся доводы в защиту важности такого подхода к архитектурному проектированию в рамках методологии ROOM, предназначенной для систем реального времени, и средства ObjectTime, поддерживающего ее. Компании ObjectTime и Rational Software Inc. начиная с 1997 года совместно работали над унификацией своих перспективных технологий, и вслед за периодом стратегической кооперации, в ходе которой ObjectTime являлась эксклюзивным поставщиком средств моделирования и технологии автоматической генерации кода компании Rational, предназначенных для систем реального времени (а Rational имела права на распространение ObjectTime Developer по всему миру), компания Rational Software объявила о приобретении компании ObjectTime в декабре 1999 года. В результате понятие описания архитектуры, впервые появившееся в методологии ROOM, теперь нашло свое применение в расширениях реального времени для UML [676]. Язык UML вместе с расширениями реального времени является, вероятно, наиболее широко распространенным и коммерчески успешным языком описания архитектуры. Он позволяет моделировать структуру системы, идентифицируя представляющие интерес ее составляющие и отношения между ними. В классе систем реального времени, описанных в рамках методологии ROOM (сложные, управляемые событиями и потенциально распределенные системы, похожие на те, которые используются в телекоммуникационных, авиакосмических и оборонных системах), в фокусе расширений реального времени оказались два типа (из девяти) общих диаграмм UML: диаграммы классов и диаграммы сотрудничества. Диаграммы классов языка UML охватывают отношения, которые являются универсальными, т.е. они применимы к любым возможным экземплярам в любых возможных контекстах. С другой стороны, диаграммы сотрудничества описывают отношения, характерные только для одного определенного контекста. Поэтому диаграммы сотрудничества позволяют различать различные случаи использования разных экземпляров, относящихся к одному и тому же классу. Эта идея отражена в понятии *роли*. Как правило, полное описание структуры сложной системы реального времени может быть получено с помощью таких расширений посредством комбинации диаграмм классов и диаграмм сотрудничества.

Чтобы в подобных системах использовать UML в качестве языка описания архитектуры, определяется три структурных компонента.

1. Капсулы (ранее известные как исполнители в методе ROOM; не нужно путать его с другим одноименным понятием, применяемым в контексте прецедентов).
2. Порты.
3. Соединители (ранее известные в методе ROOM как *связи*).

Капсула (capsule) представляет собой сложный, физический, иногда распределенный архитектурный объект, который взаимодействует со своим окружением при помощи одного или

нескольких граничных объектов, получивших название портов. Она содержит все свои порты и подкапсулы, которые не могут существовать независимо от своих капсул, "владеющих" ими (если подкапсула не является "подключаемой" капсулой), что делает ее центральным элементом моделирования в расширении UML для систем реального времени. **Порт** (port) также является физическим объектом, частью капсулы, которая реализует специальный интерфейс и играет особую роль в рамках сотрудничества между капсулами. Будучи физическими объектами, порты являются видимыми изнутри и извне капсул. Вне капсул порты можно отличить друг от друга только по их индивидуальности и той роли, которую они играют в своем протоколе. Однако если взглянуть на них изнутри капсулы, их можно причислить либо к *передающим* портам, либо к *конечным* портам. Передающие порты отличаются от конечных своими внутренними соединениями. Передающие порты подсоединяются через другие порты к подкапсулам и просто служат для передачи сигналов. В противоположность этому конечные порты подсоединены к конечному автомату капсулы и представляют собой источники и приемники сигналов. Как капсулы, так и порты моделируются в системе обозначений UML при помощи стандартных значков классов, но можно также использовать дополнительный стандартный значок — маленький черный квадрат (в противоположность белому квадрату, обозначающему парный порт в протоколе двоичной синхронной передачи данных). **Соединитель** — это физический канал связи, предоставляющий средства передачи для определенного абстрактного сигнального протокола. Соединитель может только соединять порты, играющие дополнительные роли в протоколе, с теми портами, с которыми связан данный протокол. Соединители моделируются при помощи ассоциаций языка UML. На рис. 7.1 показан наглядный синтаксис этих структурных элементов в языке UML, расширенном для систем реального времени. При таком подходе капсулы вынуждены связываться друг с другом исключительно при помощи своих портов, при этом осуществляется отделение внутреннего представления капсул от какой бы то ни было информации об их окружении и становится возможным многократное использование.

Однако здесь возникает более общий вопрос: каким образом значительно возросший формализм сам по себе будет способствовать решению фундаментальных проблем разработки программного обеспечения. Корни исследований в области архитектуры, выполненных в Университете Карнеги Меллон, связаны с поисками Мэри Шоу определения самой дисциплины программной инженерии [701]. Согласно ее утверждениям, зрелость технической дисциплины должна быть отмечена появлением "в достаточной степени научного базиса", что позволяет основной массе ученых-профессионалов использовать свою теорию как для анализа проблем, так и для синтеза решений. Прогресс очевиден, когда наука становится движущей силой. Поэтому Шоу представляет модель, согласно которой появление стройной дисциплины программной инженерии связано с интенсивным поиском соответствующей научной методологии и последующей реализацией этой методологии на практике.

Эта линейная научная модель, предшествующая практике и сводящая программную инженерию к "чистому" решению проблем, основана, конечно, на знании традиционных компьютерных наук. Но проблема заключается в том, что считать "прикладной наукой". Традиционно, обработка данных, главным образом, рассматривается как отрасль математики, поэтому эта проблема становится все более и более сложной. Например, в работе [106] отмечается, что самая интересная часть процесса создания программного обеспечения — его проектирование, использование, а также человеческий фактор: "Поскольку разработка программного обеспечения, ориентированная на человека, предполагает исследование этого феномена, можно доказать, что более правильно рассматривать ее как отрасль антропологии, а не математический метод. Исследование процесса создания программного

обеспечения в действительности может быть в значительной мере неправильно систематизировано в нынешней академической среде, что ведет к чрезмерному увлечению формальными методами и недостаточной работе по сбору сырого материала, который почти всегда поступает в анекдотической (или, по крайней мере, неколичественной) форме" (с. 36).

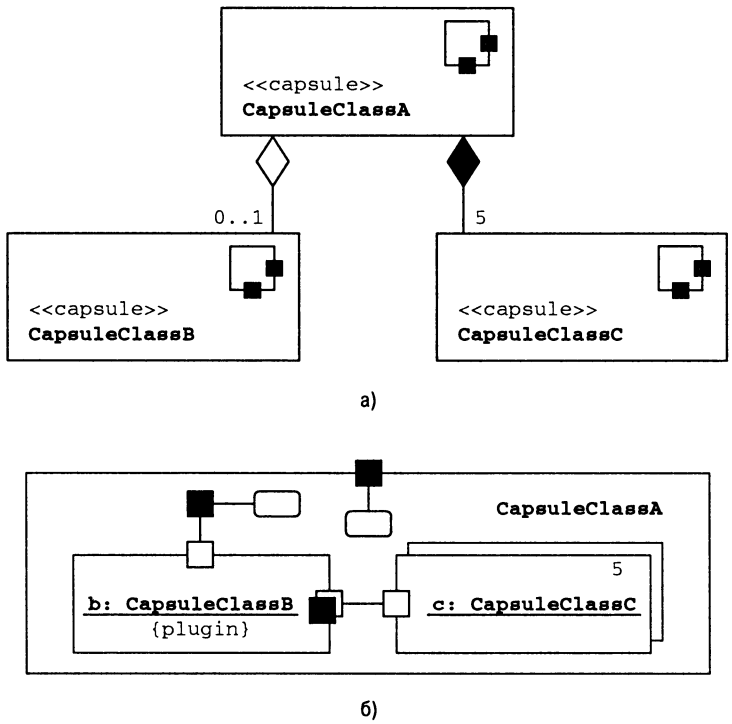


Рис. 7.1. Использование UML в качестве языка описания архитектуры (Architecture Description Language — ADL) для систем реального времени (адаптировано из работы [676]): а — диаграмма классов, выполненная в соответствии с UML-RT (UML для систем реального времени); б — одна из возможных диаграмм сотрудничества

Интересно, что Боренштейн (Borenstein) поддерживает неформальный подход и неколичественные метрики современной практики проектирования программного обеспечения. Брюс И. Блюм (Bruce I. Blum), ученый-исследователь с большим стажем, работавший в Университете Джонса Хопкинса, требовал пересмотра дисциплин компьютерных наук и программной инженерии на том же основании, которое почти совпадает с идеями Боренштейна [85, 86]. В своих работах он рассматривает образование понятий сложившейся традиции как с точки зрения философии науки, так и с точки зрения практики. Он указывает на диалектическое противоречие, проявляющееся при создании программного обеспечения, которое традиционные компьютерные науки отказываются принять. Он делает различие между "программой в компьютере" и "программой в реальном мире". Программа в компьютере представляет собой законченную математическую абстракцию и является (по крайней мере,

теоретически) проверяемой по отношению к некоторой спецификации. На самом деле основной интерес в такой программе вызывает степень сложности ее структуры. Но то, что реализовано на компьютере, должно также существовать и как "программа в реальном мире", где единственным и исключительным критерием качества является ее полезность. Ее ценность определяется способностью трансформировать ранее существовавшую ситуацию в такую ситуацию, которая не только влияет на ранее существовавшую ситуацию, но и сама испытывает влияние с ее стороны. В результате "программа в компьютере" существует в стабильной, хорошо известной и формально описываемой среде, но именно такие средства, как "программа в реальном мире", являются динамическими и не до конца понятыми. Основная проблема состоит в том, что формальные методологии представляют собой, главным образом, процесс выбора из существующего универсального множества, состоящего из надлежащим образом определенных формальных абстракций. Выбор сделать легче, чем создавать с нуля, и поэтому все эти преимущества неоценимы, но исчерпывающее решение, использующее формальные методы, возможно только тогда, когда формализованы *все возможные* проектные решения. Безусловно, институт программной инженерии заинтересован в накоплении архитектурных стилей. Однако в своих критических замечаниях Блюм высказывает предположение, что такая задача не только является сложной, но и может так и остаться нерешенной.

Рамки проблемы

Более здравый подход предложен в [411]. Здесь рамки проблемы определяются как структура, состоящая из основных частей и задачи, подлежащей решению. Основные части соответствуют уже упомянутым ранее агентам, бизнес-объектам, диалоговым окнам, действиям и прецедентам. Задача представляет собой нечто, что необходимо выполнить для удовлетворения некоторого требования, касающегося этих частей или объектов. Затем автор абстрагируется от объектов и обращается к предметным областям и явлениям, которые являются для них общими, — с точки зрения элементов, которые могут быть описаны при помощи языков, характерных для двух областей. Еще более важно то, что рамки каждой проблемы могут обладать набором правил, связывающих пары областей. Мы полагаем, что рамки проблемы предоставляют не только способ реализации инженерии требований, но и потенциально архитектурный подход, поскольку они описывают не решение, а соответствующий метод его нахождения. Кроме того, они предусматривают использование шаблонов, которые мы рассмотрим в следующем разделе. Идея в большей степени фокусирует внимание на требованиях аналитика, пытающегося понять проблему и выбрать подход, нежели на требованиях проектировщика, который уже выбрал архитектурный стиль и технологию реализации.

Типичные рамки проблемы содержат следующие составляющие.

- Соединение: вводит отдельную предметную область между областями применения и решения. Примеры: почтовое отделение; технология CORBA.
- JSP: помогают описывать программу с точки зрения ее входных и выходных потоков. Пример: типичная система учета, отчетности и контроля запасов.
- Простое управление: описывает ситуацию, где известные управляющие правила применяются к явлениям, подлежащим управлению. Примеры: встроенные контроллеры реального времени; торговые автоматы.

- Простые информационные системы: аналог реального мира для JSP-фрейма; проблема касается пользователей, запрашивающих и обновляющих информацию, относящуюся к некоторой области. Пример: системы баз данных.
- Заготовки: описывает характерные команды, предназначенные для манипуляций над объектами. Пример: текстовые редакторы.

Рамки проблемы, взятые вместе, представляют язык шаблонов, поскольку реальные проблемы, как правило, включают несколько рамок проблем. Джексон утверждает, что идентификация рамок проблемы должна предшествовать выбору соответствующей методологии. В качестве такого метода он приводит пример анализа прецедентов и отмечает некоторые из его ограничений (что мы увидим в главе 8) с целью продемонстрировать, что использование анализа ограничено рамками проблемы, где доминирует пользовательский ввод-вывод. Все это выглядит достаточно правдоподобно, но моделирование объектов как некой формы представления знаний в какой-то мере позволяет обойти эти ограничения. Хотя объектная технология, несомненно, не свободна от ограничений, зависящих от рамок проблемы, она может быть настроена таким образом, чтобы работать с достаточно разными рамками проблем. Это объясняется большими семантическими возможностями, предоставляемыми наборами правил, а также, например, включением условий инвариантности, а также пред- и постусловий.

Мы посоветовали бы читателям выделить собственный набор хорошо знакомых проблем и выяснить, существуют ли на этом уровне обобщения другие рамки проблем. Описание этих рамок при помощи языка Джексона заинтересованному читателю предлагается сделать самостоятельно в виде упражнения.

Рассматривая аналогичные вопросы, авторы работ [188, 602] по разным причинам утверждали, что для получения всех преимуществ объектной технологии она должна использоваться для моделирования процессов в двух концептуально отличных друг от друга пространствах: в предметной области, где создается концептуальная модель ситуации, существующей в реальном мире, и в пространстве решений, где разрабатывается программная часть решения. Типы объектов охватывают наблюдаемое поведение ключевых абстракций в предметной области, но определяют поведение проектного решения в пространстве реализаций. Программист управляет последним, но никогда не сможет сделать то же самое с первым. Несмотря на одинаковое использование объекта в качестве базового модельного представления в обоих пространствах, фундаментальное различие состоит в природе самих этих пространств, а следовательно, и моделируемых в них абстракций. Это делает переход из одного пространства в другое нетривиальной проблемой. Автор настаивает на необходимости использования сущности, похожей на двуликого Януса, которая находится между этими двумя пространствами и смотрит в обоих направлениях одновременно. Именно это он называет архитектурой программного обеспечения. Самое интересное, что хотя Боренштейн, Блюм и О'Каллаган (к которым можно также присоединить голоса Митча Кэпора [433] и Терри Винограда [799]), как оказалось, придерживаются представления, противоположного современному доминирующему представлению об архитектуре программного обеспечения, их взгляды действительно имеют много общего с историческими корнями понятия архитектуры программного обеспечения.

Фред Брукс мл. (Fred Brooks Jr.) в своей книге [118] утверждал, что самым важным при проектировании системы является ее концептуальная целостность. Для него это являлось определяющим свойством программного обеспечения или архитектуры системы, а также основной задачей архитектора. Брукс утверждал, что гораздо лучше поддерживать связанный

набор идей проектирования и в случае необходимости опускать характеристику или функциональные возможности с целью поддержания этой связанности, чем создавать систему, содержащую разнородные или несогласованные понятия, даже если каждое из них, рассмотренное отдельно, может быть совсем неплохим. В этом случае нужно задать себе такие вопросы.

- Каким образом можно достичь концептуальной целостности?
- Можно ли поддерживать концептуальную целостность, не делая при этом различия между небольшой "архитектурной" элитой и многочисленной группой "рядовых" конструкторов системы?
- Каким образом воспрепятствовать тому, чтобы архитекторы не предлагали нереализуемых или сверхдорогих реализаций?
- Каким образом можно добиться такого положения дел, чтобы архитектура была отражена в детальном проектном решении?

Отвечая на эти вопросы, Брукс сознательно обращается к модельному представлению архитектуры в среде построения системы. Он делает строгое разграничение между архитектурой и реализацией. Подобно Блаауву (Blaauw), он приводит простой пример часов, архитектура которых включает циферблат, стрелки и кнопку завода. Изучив архитектуру часов, ребенок для определения времени может воспользоваться как наручными, кухонными, так и часами на башне. Механизм, посредством которого представляется функция, сообщающая время, является деталью реализации и исполнения, но не архитектуры.

"При помощи архитектуры системы, — говорит Брукс, — я имею представление о полной и подробной спецификации пользовательского интерфейса. Для компьютера — это руководство по программированию, а для компилятора — руководство по языку. Для управляющей программы — это руководства по языку или языкам, используемые для вызова ее функций. Для всей системы в целом — это комплект руководств, предназначенных для пользователя и оказывающих ему помощь при выполнении всей его работы." (с. 45)

Хотя это определение кажется сегодня не вполне адекватным, оно имеет, по крайней мере, одно преимущество: устанавливает тот факт, что архитектор является агентом клиента, но не разработчиков. Это, безусловно, означает, что концептуальная целостность системы, представленная архитектурой, приобретает свои очертания благодаря представлению требований клиента и что окончательным критерием "хорошей" архитектуры является ее полезность. Рассматривая этот важный аспект, нельзя не упомянуть о строгой согласованности между идеями Брукса и Блюма (хотя ироничный Блюм не использует термин "архитектура программного обеспечения") и разрыве между подходами Брукса и института SEI. Кроме того, следует отметить, что подобное представление архитектора программной системы как клиентского агента характерно также для недавно организованного Всемирного института архитекторов программного обеспечения [808].

Как и Блааув, Брукс высказывает предположение, что вся созидательная работа состоит из трех этапов: создания архитектуры, реализации и выполнения. Согласно Бруксу, архитектура появляется в конце разработки вместе с полной спецификацией системы; проектирование "модулей, табличных структур, разбиение на этапы и фазы, алгоритмы и инструментальные средства всех видов" относятся к реализации. Брукс полагает, что все три этапа могут существовать до некоторой степени параллельно и что успешное проектное решение требует постоянного диалога между архитекторами и специалистами, выполняющими реализацию.

Тем не менее участие архитекторов в таком диалоге ограничивается внешним описанием системы. Здесь налицо некоторое противоречие, которое существует между подобной идеей, определяющей роль архитектора, и настоятельным требованием Брукса поддерживать концептуальную целостность системы. По крайней мере, в современных системах, в которых учтены проблемы масштабирования и распределенности, не существовавшие в 1975 году, обратное проектирование (восстановление описания и требований по программной реализации) является самым важным фактором поддержки целостности системы.

Архитектура как логическое обоснование проектного решения

В последнее время появилась необходимость в объединении понятия архитектора как клиентского агента, отвечающего за концептуальную целостность системы, с понятиями, затрагивающими внутреннюю структуру системы. Многие современные теоретики архитектуры программного обеспечения уходят корнями к конструктивной статье [623], в которой архитектура программного обеспечения определяется в следующем виде.

Архитектура ПО = {*Элементы, Форма, Логическое обоснование*}

Бэри Боэм (Barry Boehm) уточнил последний термин *Логическое обоснование* и предложил использовать *Логическое обоснование/ограничения*. Интерпретация этой идеи, оказавшая большое влияние на развитие объектно-ориентированного подхода, была предложена Филиппом Крачтеном (Phillipe Kruchten) в работе [462], посвященной вопросам архитектуры программного обеспечения. Он поддерживает унифицированный процесс разработки и рассматривает архитектуру в качестве ее центрального компонента. Крачтен признает, что архитектура программного обеспечения связана с абстракцией, композицией и декомпозицией, а также со стилем и эстетикой. Чтобы рассмотреть все эти аспекты, особенно с учетом крупных и перспективных систем, Крачтен предлагает общую модель, состоящую из пяти следующих представлений.

- *Логическое представление* — объектная модель этапа проектирования.
- *Представление процессов*. Моделирует распараллеливание и синхронизацию проектного решения.
- *Физическое представление* — модель установления соответствия между программным обеспечением и элементами аппаратных средств, включая вопросы распределения.
- *Представление разработки*. Статическая организация программного обеспечения в рамках соответствующей среды разработки.
- *Сценарное представление* — сценарии использования, благодаря которым отчасти определяется архитектура и проверяется ее правильность.

Крачтен применяет уравнение Перри (Perry) и Вульфа (Wolf) отдельно для каждого представления. Для каждого представления определяется набор компонентов, контейнеров и соединителей, а сами представления используются в качестве форм и шаблонов, работающих в определенном контексте архитектуры. Аналогичным образом для каждого представления задается также логическое обоснование и ограничения, что связывает таким образом архитектуру

с требованиями. Каждое представление отображается в виде схемы с использованием системы обозначений, соответствующей данному представлению (в оригинальной статье, которая появилась раньше UML, для каждого представления применялись подмножества системы обозначений Буча (Booch)).

Критикуя предложенный Виттом [805], довольно незатейливый четырехступенчатый (построение, систематизация, определение и оптимизация) процесс разработки архитектуры, включающий двенадцать стадий, Крачтен предлагает итеративный подход, управляемый сценариями. На каждой итерации, исходя из относительного риска и критичности, выбирается небольшое количество сценариев. Затем создается предварительная архитектура. С целью получения основных абстракций (классов, диаграмм сотрудничества, процессов, подсистем и т.д.) и последующего разбиения на пары "объект-операция" пишутся сценарии. Выявленные архитектурные элементы распределяются по четырем представлениям: логическому, физическому, описывающему процесс и разработку. Затем архитектура реализуется, тестируется, оценивается и анализируется, при этом возможно обнаружение ошибок или выявление возможностей для усовершенствования. На этом этапе могут начаться последующие итерации. Документация, полученная в результате этого процесса, на самом деле представляет собой два документа: *документ по архитектуре программного обеспечения* (рекомендуемый для него шаблон приведен в табл. 7.1) и отдельные *рекомендации по проектированию программного обеспечения*, содержащие наиболее важные проектные решения, которые должны быть учтены с целью поддержания концептуальной целостности архитектуры. Суть процесса заключается в том, что первоначальный прототип архитектуры развивается в процессе создания окончательной версии системы.

Проницательные читатели обнаружат в этом подходе признаки описанного в книге [416] унифицированного процесса (Unified Process), который представляет собой бесплатно распространяемую версию запатентованного процесса RUP (Rational Unified Process), а также результаты OMG (рабочей группы по развитию стандартов объектного программирования) по созданию стандартного процесса объектно-ориентированной разработки. В современном представлении, включающем систему обозначений UML, модель Крачтена "4 +1" была сохранена, но с несколько измененной терминологией. Представления процессов, разработки и логическое представление остались, физическое представление теперь называется представлением реализации (Implementation View) и/или представлением компонентов (Component View) (в настоящее время Гради Буч использует эти названия в рамках одного и того же представления как взаимозаменяемые), а сценарное представление именуется как представление прецедентов (Use Case View), прочно связывая модель "4+1" с UML и унифицированным процессом. В своей программной речи, обращенной к сообществу приверженцев UML, Буч пошел еще дальше и определил архитектурную метамодель, которая изображена на рис. 7.2 (см. работу [100]). Мы вернемся к ней позднее, но здесь можно отметить, что метамодель определяет следующее. *Архитектура программного обеспечения* задается при помощи ее *описания*, состоящего из *архитектурных представлений* (*логического*, *процессов*, *реализации*, *развертывания* и *прецедентов*), причем каждое из них отображается на *архитектурной схеме* и в *руководстве по архитектурному стилю*. Там же показано бинарное отношение между *архитектурой программного обеспечения* и *требованиями*. Принимая во внимание переименование оригинального *Руководства по проектированию программного обеспечения* в *Руководство по архитектурному стилю*, нужно отметить полную совместимость метамодели с моделью "4+1".

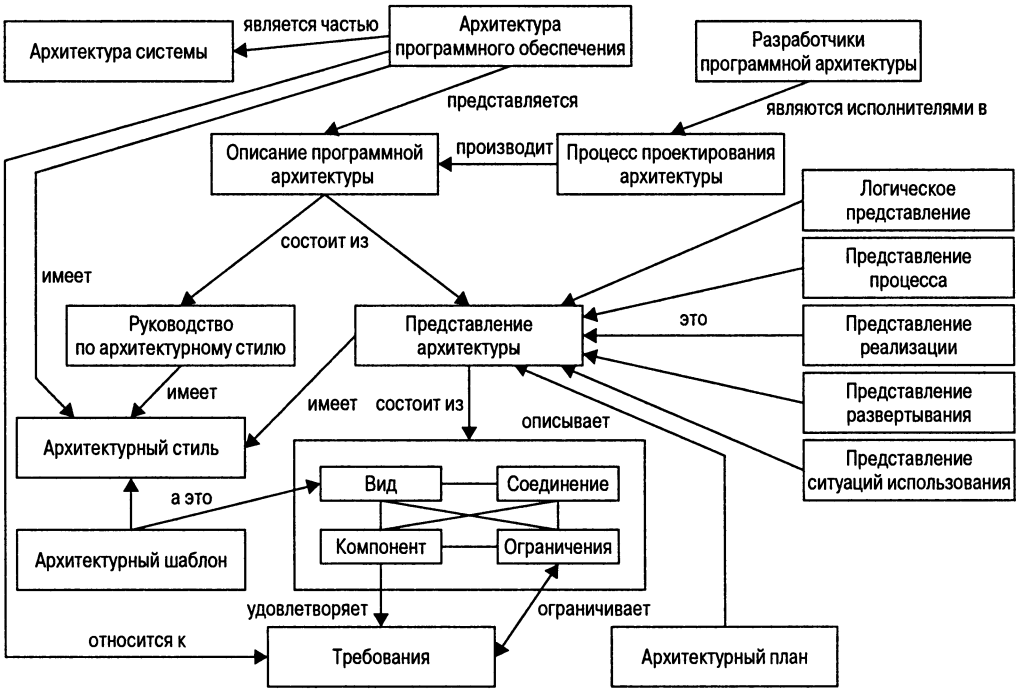


Рис. 7.2. Метамодель архитектуры программного обеспечения, предложенная Бучем (по материалам работы [100])

Могут возникнуть небольшие сомнения относительно того, что модель "4+1" и ее преемники вносят значительный вклад в разработку архитектуры программного обеспечения как практическую дисциплину. Если сравнивать ее с редукционистским представлением архитектуры программного обеспечения, упомянутый выше вклад реализуется двумя конкретными способами. Во-первых, модель "4+1" расширяет диапазон архитектуры, не ограничиваясь исключительно структурой. В известном выступлении Буча один слайд, посвященный области архитектуры программного обеспечения, описывает ее не только при помощи традиционной терминологии, предполагающей ответы на вопросы *что* и *как*, а также *почему* и *кто*. В работе [101] предлагается определение, которое, помимо массы других полезных свойств, предполагает, что "архитектура программного обеспечения затрагивает не только структуру и поведение, но также использование, функциональные возможности, производительность, устойчивость, многократное использование, возможность понимания, экономические и технологические ограничения и компромиссные решения, а также вопросы эстетики" (с. 458). Во-вторых, она определяет путь разработки программ, поскольку первый прототип должен быть архитектурным. В унифицированном процессе разработки программного обеспечения (см. работу [416]) такой прототип называется "маленькой тощенькой системой".

Шаблон описания архитектуры программного обеспечения, предложенный Крачтеном в 1995 году, включает следующее.

Титульная страница
История изменений
Оглавление
Список рисунков

- Аннотация
- Ссылки
- Программная архитектура
- Архитектурные цели и ограничения
- Логическая архитектура
- Архитектура процессов
- Архитектура разработки
- Физическая архитектура
- Сценарии
- Размер и производительность
- Качество

Приложения

- Список сокращений и условных обозначений
- Определения
- Принципы проектирования

В модели "4 + 1" и унифицированном процессе архитектура представлена как часть самого процесса разработки программного обеспечения. В своей оригинальной статье Крачтен показывает, как этот процесс реализуется в таких особых областях, как архитектура Telic PABX (локальная АТС с исходящей и входящей связью с городом) и система управления движением воздушного транспорта. Однако не совсем понятно, как разработать архитектуру для другой предметной области, семейства систем или серийной системы, каждая из которых должна отталкиваться от общей архитектуры. Как-никак, если каждое проектное решение требует создания своей архитектуры, возрастает вероятность (что весьма очевидно) того, что каждая такая "архитектура" будет не похожа на другие.

Ян Бош (Ian Bosch) недавно изложил свои взгляды на проблему развития подхода к созданию серийных продуктов. Его взгляды основываются на работе группы RISE (Research in Software Engineering), выполняющей исследования в области программной инженерии в Университете Карлскрона/Роннеби (Швеция), и на результатах ее сотрудничества с такими компаниями, как Securitas Alarm AB и Axis Communications [108]. Подходы, предусматривающие серийное производство продуктов, весьма близки технологиям компонентной разработки. Подход Боша предусматривает вначале разработку так называемого "архитектурного проектного решения, основанного на функциональных возможностях", которое берет начало

от спецификации требований. Суть этого вида деятельности состоит в поиске ключевых архитектурных абстракций для данной проблемы, которые автор называет "архетипами". Архетипы представляют собой типы объектов, корни которых лежат в предметной области[1], но при разработке серийных продуктов эти типы, как правило, подвергаются дальнейшему обобщению и абстрагированию, выполняемому на основе конкретных ситуаций, с которыми впервые сталкивается аналитик или архитектор. Далее Бош утверждает, что многие архетипы по своему характеру фактически являются универсальными (независимыми от предметной области).

Небольшой и постоянный набор архетипов создается из более обширного списка кандидатов, в котором первоначальные кандидаты часто объединяются друг с другом с целью создания абстракций более высокого уровня. Затем идентифицируются и выбираются абстрактные отношения между этими архетипами. Далее формируется структура программной архитектуры посредством рекурсивного разбиения набора архетипов на определенные отношения между ними. На следующем этапе для подтверждения правильности полученных результатов может быть описан процесс *инстанцирования системы*, что требует дальнейшего разбиения на компоненты более низкого уровня и заполнения этих компонентов инстанцированными архетипами. Поскольку архитектуры, предназначенные для серийных продуктов, должны поддерживать самые разнообразные системы, обладающие не только разными описаниями, но и различными реализациями, изменчивость должна учитываться уже на этой ранней стадии. Поэтому инстанцирование такого рода выполняется с целью подтверждения соответствия архитектуры требованиям.

При оценке архитектуры применяется другая достаточно оригинальная идея *профилей*. Профиль представляет собой набор сценариев, которые не обязательно являются прецедентами. Даже в случае применения прецедентов для проверки соответствия функциональным требованиям, они (прецеденты) представляют лишь подмножество полного перечня описанных сценариев. Кроме них, задаются другие сценарии, связанные с атрибутами качества (нефункциональными требованиями), например *сценарии риска* (для тех систем, в которых безопасность имеет первостепенное значение) или *сценарии изменений* для гибкой поддержки. Профили предусматривают наличие более точного описания атрибутов качества по сравнению с типичным описанием, которое чаще всего применяется при разработке систем. Бош отмечает, что обычно используют от 4 до 8 категорий сценариев. Сценарии идентифицируются и определяются для каждой категории с учетом веса. При необходимости веса нормализуются. Процесс получения оценки, основанной на сценариях, включает две основные стадии: анализ влияния и прогнозирование атрибутов качества. Оценивается влияние выполнения каждого сценария и затем вычисляется прогнозируемое значение атрибута качества. Так, например, ряд измененных и новых компонентов, полученных в результате сценария изменения, относящегося к категории поддержки, может на второй стадии привести к повышению расходов на сопровождение. Профили и сценарии могут использоваться самым различным образом, включая вычисление оценок для самой системы и динамическую проверку архитектуры.

В ходе вычисления этих оценок может оказаться, что архитектура не удовлетворяет одному или нескольким требованиям качества, как правило, в результате противоборствующих сил, воздействующих на весь проект в целом. Например, ограничения, накладываемые на

---

[1] Это понятие, как оказалось, похоже на понятие, описанное в шаблоне Archetype (Архетип) языка шаблонов ADAPTOR [602]. Однако данное понятие отличается от соответствующего понятия Питера Коада (Peter Coad), использующего этот термин для описания одного из четырех видов сотрудничества на метауровне, которое, как он полагает, должно быть эндемическим в объектных системах [168].

рабочие характеристики, будут нарушены ввиду наличия определенных уровней перенаправления, которые были предусмотрены в архитектуре в целях удовлетворения требований изменчивости. В таких случаях требуется то, что Бош называет "архитектурными трансформациями". Можно выделить четыре категории архитектурных трансформаций, которые применяются в следующем порядке убывания их предполагаемого влияния.

- Применить архитектурный стиль (согласно цитируемой работе Шоу и Гарлана).
- Применить архитектурный шаблон (или механизм). Под этим Бош подразумевает локальное применение правила с целью определения, каким образом в системе будет обеспечен один из аспектов ее полной функциональности, например параллельность или перманентность[2].
- Применить шаблон проектирования (согласно [290], см. ниже).
- Преобразовать качественные требования в функциональные — например, дополнить функциональные возможности самоконтролем или предусмотреть дублирование для повышения отказоустойчивости.

На завершающей стадии выполняется распределение качественных требований по идентифицированным компонентам или подсистемам, относящимся ко всей архитектуре в целом. Мы ограничились рассмотрением лишь той части книги Боша, которая посвящена архитектуре программного обеспечения. Вторая (и основная) часть посвящена реальной разработке серийных продуктов. Заинтересованным читателям рекомендуем приобрести саму книгу.

На самом деле при разработке серийных продуктов, между представлением "4+1" и подходом к программной архитектуре, который используется исследовательской группой RISE, можно обнаружить довольно много общего. В этих работах уделяется должное внимание всей области программной архитектуры, в том числе концептуальной целостности, логическому обоснованию и структуре[3]. При этом связываются нефункциональные требования с компонентами или подсистемами, первоначально выделенными посредством анализа функциональности. Авторы рекомендуют на определенной стадии процесса применять архитектурные стили и шаблоны проектирования. Мы рассмотрим все эти вопросы ниже. Основное различие состоит в том, что методология Крачтена является восходящей и начинается с абстракций, выявленных в спецификации требований к системе, тогда как подход Боша, несомненно, является нисходящим — он начинается с архитектурных абстракций и предполагает их последующее применение для разработки конкретного приложения или системы.

---

[2] Это понятие, безусловно, отличается от приведенного в книге [133] и рассматриваемого далее в этой главе.

[3] Следует отметить, что на заседании экспертов по вопросу "Что такое программная архитектура" на конференции TOOLS Europe 2000, на которой среди всех остальных присутствовали два соавтора этой главы, Бош без колебаний причислил Крачтена к "структуралистскому" лагерю специалистов, занимающихся вопросами архитектуры программного обеспечения.

7.2. Шаблоны, архитектура и раздельное проектирование

Одной из наиболее важных современных идей разработки программного обеспечения является идея шаблонов проектирования. Шаблоны проектирования представляют собой стандартные решения периодически появляющихся проблем. Шаблонам даны имена с той целью, чтобы облегчить специалистам их обсуждение, а также обдумывание проектного решения. Они всегда были популярны при компьютерной реализации вычислений, поэтому такие термины, как "связный список" или "рекурсивный спуск", понятны специалистам, работающим в этой области.

Шаблоны программного обеспечения были описаны как многократно используемые микроархитектуры. Шаблоны представляют собой абстрактные базовые решения проблем, которые могут неоднократно возникать в различных контекстах. Конкретная реализация решения в разных приложениях может быть различной. Поэтому шаблоны нельзя рассматривать как готовые "встраиваемые" решения. Обычно в объектно-ориентированной разработке они представляются в виде повторяющихся схем классов, а также структурных и динамических соединений между ними. Вероятно, наиболее известные и успешно применяемые примеры шаблонов можно найти в каркасах приложений, связанных с построением графических пользовательских интерфейсов или с другими хорошо известными проблемами разработки. В действительности появление шаблонов отчасти связано с представлением уже существующих каркасов приложений: специалистам захотелось знать, насколько общим является использованный подход. В настоящее время более распространена поставка каркасов приложений в форме гибких библиотек классов, предназначенных для использования программистами, применяющими такие языки, которые поддерживают понятие класса. Чаще всего это языки C++ и Java. Диапазон примеров каркасов приложений достаточно велик — от библиотек классов, поставляемых вместе со средой программирования через NeXtStep Interface Builder, до многочисленных графических пользовательских интерфейсов и систем разработки клиент/сервер, которые можно найти на рынке, например Delphi, Visual Studio, Visual Age и Visual Basic.

Шаблоны являются наиболее удобным средством, так как они предоставляют проектировщикам язык общения. Они позволяют объяснить сложную идею, не начиная при этом с нуля: проектировщик может просто назвать шаблон по имени, и все будут знать, по крайней мере в общих чертах, о чем идет речь. Именно таким образом проектировщики, работающие в различных предметных областях, сообщают о своих идеях проектирования. В этом смысле шаблоны являются превосходным средством накопления и распространения неколичественных данных, которые, согласно утверждениям Боренштейна [106], должны быть собраны еще до того, как мы сможем увидеть реальные достижения в процессе создания программного обеспечения. Как и в случае с архитектурой программного обеспечения, существует два различных способа представления шаблонов, и оба они обладают достаточной ценностью. В целях исследования шаблонов рассмотрим сначала корни самого понятия, которые относятся не к сфере разработки программного обеспечения, а к области его построения. Тогда вряд ли читателя удивит тот факт, что шаблоны тесно связаны с архитектурой программного обеспечения.

Шаблоны связывают с радикальными идеями архитектора Кристофера Александера (Christopher Alexander), работавшего в области строительства. Уже в самом начале своей карьеры он пришел к мысли, что подавляющее большинство сооружений, построенных с

момента окончания второй мировой войны (и составляющих основу всех работ по строительству, выполненных человеком на протяжении всей истории его существования и развития видов), было антигуманным ввиду плохого качества и полного отсутствия чувства красоты. В своей первой работе Александер сильно критиковал современное проектирование (см. работу [21]), противопоставляя провалы профессионального проектирования, *обладающего самосознанием*, тому, что он называл *бессознательным* процессом, при помощи которого создавались крестьянские дома, хижины эскимосов и лачуги племен в Камеруне. В последнем случае "... шаблон строительства, шаблон поддержки строения, ограничения, накладываемые на окружающие условия, а также шаблон повседневного использования сливаются воедино" (с. 31), и даже отсутствует понятие "проектного решения" или "архитектуры", не говоря уж о разделении специалистов на разработчиков и архитекторов. Каждый человек строит свой собственный дом.

Александер утверждает, что бессознательный процесс обладает гомеостатической (т.е. самоорганизующейся) структурой, которая позволяет получать весьма подходящие формы даже в случае изменений. Однако в рамках процесса, обладающего самосознанием, такая гомеостатическая структура разрушалась, неминуемо приводя к плохо согласованным формам.[4] Хотя по определению нет явных четко сформулированных правил строительства, использующего бессознательный процесс, тем не менее обычно существуют многочисленные невысказанные, неписаные и неявные правила, неукоснительно поддерживаемые культурой и традицией. Эти традиции образуют основу стабильности и еще в большей степени сопротивляются всем изменениям, в том числе и самым необходимым — как правило, в том случае, когда форма некоторым образом "разваливается". Когда такие изменения становятся необходимыми, именно простота самой жизни и безотлагательность обратной связи (поскольку строитель и домовладелец зачастую — это одно и то же лицо) означают, что необходимая переделка может быть сама по себе одноразовой. Таким образом, бессознательный процесс характеризуется быстрым реагированием на разрозненные "поломки", сочетающимся с сопротивляемостью всем остальным изменениям, что позволяет в случае применения этого процесса выполнять ряд незначительных последовательных поправок вместо глобальных скачкообразных исправлений. Изменения оказывают только локальное воздействие, и на протяжении длительного периода времени система регулирует "подсистему подсистемой". Поскольку второстепенные изменения происходят быстрее культурных, баланс остается неизменным и динамически переустанавливается после каждого возмущения.

При формализованном процессе разработки влияние традиции не имеет такой силы или вообще отсутствует. Цепочка обратной связи удлиняется ввиду наличия дистанции между "пользователем" и строителем. Немедленная реакция на нарушение невозможна, поскольку материалы не находятся "под рукой". Поэтому нарушения накапливаются и нуждаются в

---

[4] Развитые биологические системы являются гомеостатическими. Рассмотрим, как формируется дерево, например древесина могучего дуба. Форма конкретного дерева, как оказалось, хорошо согласуется с его окружением. Высоту дерева можно рассматривать как результат его конкуренции с близлежащими деревьями. Если деревья используются для защиты от ветра и расположены на границах крестьянских хозяйств, то они обычно согнуты в том направлении, которое задается шаблонами преобладающей розы ветров. Количество ветвей дерева зависит от числа листьев, которые оно выпускает в соответствии с местными условиями, определяемыми солнечным освещением, количеством осадков и т.д. Если дерево одиноко растет на вершине холма, шаблон роста, как правило, симметричен. Однако если существуют какие-либо ограничения, они отражены в собственном шаблоне роста дерева. Приспособляемость дерева является, конечно, функцией его генетического кода. Позднее Александер говорил о своем подходе как о "генетическом", и задача шаблонов заключается в том, чтобы постепенно внедрять этот генетический код в структуры.

более радикальном действии, поскольку они должны обрабатываться все вместе. Упомянутые выше факторы, приводящие процесс конструирования в состояние равновесия, в формализованном процессе исчезают. Равновесие, если оно вообще достигается, не может быть поддержано даже в минимальной степени, так как темпы культурных изменений опережают скорость возможных исправлений.

Александер не видит возможности возврата к примитивным формам и предпочитает рассматривать новый подход к современной дилемме. "Обладающие самосознанием" проектировщики и, несомненно, само понятие проектирования появляются в результате возросшей сложности требований и совершенствования материалов. В настоящее время они контролируют процесс до такой степени, какой бессознательный специалист никогда не достигал. Но чем больше контроля они получают, тем значительнее бремя знаний, тем больше они затрачивают усилий, пытаясь объять все, и тем более непонятной становится структура проблемы, выражающая причинно-следственные связи. Такая структура должна быть создана с целью выработки правильного решения.

В 1964 году в своей работе Александер делает попытку решить обозначенные им проблемы и описывает полуалгоритмический, механистический "план действий", основанный на функциональной декомпозиции (поддерживаемой математическим описанием, изложенным в приложении). Позднее он отказался от своих результатов. Эти рисунки, больше похожие на неформальные, он использовал в рабочем примере, и они, как оказалось, обладают высокой значимостью. Случилось так, что они стали основой для шаблонов в его более поздней работе.

В настоящее время "теория" Александера имеет вид литературного проектного решения объемом в 11 томов, причем это проектное решение не включает его работу 1964 года. На сегодня восемь из этих томов опубликованы (хотя, в лучшем случае, известны три из них). Многие с нетерпением ожидают появления девятого тома, поскольку предполагается, что в нем будет содержаться самое полное описание теории, лежащей в основе шаблонов. Общей темой для всех этих книг является отказ от абстрактных категорий архитектурных принципов или случайных принципов проектирования. Кроме того, отбрасывается идея возможности успешного проектирования "очень абстрактных форм на высоком уровне" ([24], с. 8). Согласно Александеру, архитектура достигает своего наивысшего выражения не на уровне общей структуры, а при отображении ее мельчайших деталей, что он называет "тонкой структурой". Иными словами, понятность проектного решения на макроуровне может быть достигнута в результате согласованности решений на более низких уровнях — геометрическое единство должно поддерживаться на всех уровнях масштаба. Один ум не может представить в своем воображении всех уровней этой рекурсивной структуры до ее реального построения. И в этом контексте подразумевается использование шаблонов, предназначенных для строительства.

В работе [23] представлен язык архитектурных шаблонов, используемых в строительстве. Этот язык представляет собой набор из 253 взаимосвязанных шаблонов, в которых инкапсулированы наилучшие практические проектные решения самых разных уровней: от размещения ниш до строительства городов и административных центров. Язык предназначен не только для использования разработчиками, но и для совместного применения всеми организаторами работ. В качестве предпосылки (отчасти, по крайней мере) может рассматриваться идея, согласно которой эксперты по зданиям — это те специалисты, которые живут и работают в них, а не те, которые формально изучили архитектуру или строительное инженерное искусство. Шаблоны последовательно применяются непосредственно в самом строительстве. Каждое изменение состояния, вызванное применением шаблона, создает новый контекст, в котором может быть применен следующий шаблон. И вся разработка целиком представляет собой результат последовательного применения языка шаблонов. Поэтому

данный язык имеет созидательный характер: он позволяет генерировать проект постепенно на основе решения каждой отдельной проблемы, решаемой каждым из шаблонов.

Waist-High Shelf (полка на уровне пояса) (номер 201 в языке) представляет собой типичный шаблон. Он предлагает строить полки на уровне пояса по всем основным помещениям с целью поддержания такого "местоположения" объектов, чтобы они всегда находились под рукой. Очевидно, что конкретная форма, глубина, расположение и другие характеристики этих полок будут различными в разных домах и на разных рабочих местах. Поэтому реализация шаблона создает весьма специальный контекст, в котором могут быть использованы другие шаблоны, например шаблон Thickening The Outer Wall (утолщение наружной стены) (номер 211), поскольку Александер полагает, что полки должны встраиваться, где это необходимо, в саму структуру здания; для заполнения полок применяется шаблон Things From Your Life (вещи вашей жизни) (номер 253).

Однако шаблон Gradual Stiffening (постепенное укрепление), в большей степени, чем любой другой, является физическим и методологическим воплощением подхода Александера к проектированию (номер 208).

Основная философия, лежащая в основе использования языков шаблонов, заключается в том, что строения должны быть уникальным образом приспособлены к индивидуальным требованиям и местам расположения и что планировки зданий должны быть свободными и изменяемыми с целью приспособления всех деталей.

Надо признать, что здание не собирается из компонентов некоторого комплекта. Конструкция сплетается, с самого начала являясь глобально завершенной, но хрупкой. Затем она постепенно становится все более и более жесткой, но все еще достаточно непрочной, и только в конце она полностью закрепляется и становится прочной ([23], с. 963–969).

Описывая этот шаблон, Александер предлагает читателю мысленно представить работу пятидесятилетнего квалифицированного столяра. Он выполняет работу без остановки до тех пор, пока наконец не создаст качественное изделие. Он делает свою работу постепенно и монотонно, разделяя ее на небольшие последовательные инкрементные шаги, которые позволяют ему всегда исправить ошибку или устранить дефект в ходе последующего шага. Александер сравнивает его действия с работой неопытного специалиста, который с "паническим вниманием к деталям" пытается решить все вопросы заранее, испытывая страх перед тем, что он сделает непоправимую ошибку. Александер обращает внимание на то, что почти вся современная архитектура по своему характеру напоминает действия неопытного специалиста, но никак не работу квалифицированного мастера. Успешные процессы строительства, позволяющие создавать надлежащим образом приспособленные формы, предусматривают отсрочку решений относительно деталей проекта до выполнения самого процесса строительства с тем, чтобы такие детали подходили ко всей развертывающейся конструкции целиком.

Как оказалось, идеи Александера впервые были внедрены в сообществе специалистов по объектно-ориентированным технологиям Кентом Беком (Kent Beck) и Вардом Каннингамом (Ward Cunningham). В статье журнала Smalltalk Report, датированной 1993 годом, Бек объявил об использовании шаблонов уже на протяжении шести лет, однако движение в поддержку шаблонов программного обеспечения началось с семинара, посвященного выпуску справочного руководства для разработчиков архитектуры программного обеспечения; этот семинар был организован Брюсом Андерсоном (Bruce Anderson) для конференции OOPSLA'91 (Object-oriented programming, Languages & Applications — Объектно-ориентированное программирование, языки и приложения). Здесь впервые встретились Эрик Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides) — компания, впоследствии получившая название Группы Четырех (Gang of

Four — GoF). Гамма уже был близок к завершению своей кандидатской диссертации, посвященной "шаблонам проектирования" в рамках каркаса ET++ [290]. Он уже объединился с Нелмом, совместно с которым создал независимый каталог. Ко времени новой встречи на конференции OOPSLA в 1992 году вначале Влиссидес, а затем и Джонсон присоединились к работе, и где-то в 1993 году группа пришла к мысли о написании книги, которая стала бестселлером с момента ее публикации в 1995 году. В действительности же за рамками самого движения в поддержку шаблонов многие специалисты, работающие в области разработки программного обеспечения, отождествляли шаблоны программного обеспечения целиком и полностью с книгой Группы Четырех.

Но конференция OOPSLA 1991 года была только первой в серии встреч, которые увенчались вначале созданием некоммерческой группы Hillside Group[5] (очевидно, названной так потому, что они отправились в один из уикендов на склон холма для апробирования строительных шаблонов Александера), и впоследствии первой конференцией, посвященной языкам шаблонов программирования (PLoP), прошедшей в 1994 году. Конференции PLoP, которые организовывала и финансировала группа Hillside Group, проходили ежегодно в Америке, Германии и Австралии, а созданные коллекции шаблонов публиковались в издательстве Addison-Wesley — к моменту написания этой книги вышло уже четыре тома. Кроме того, группа Hillside Group поддерживает Web-узел и многочисленные списки рассылки шаблонов [387]. Такие коммуникационные каналы являются основой большого и все расширяющегося сообщества, которое справедливо называется движением в поддержку шаблонов.

Публикация шаблонов сопровождалась проведением так называемых семинаров авторов шаблонов, в рамках которых проводилось рецензирование шаблонов, немного напоминающее рецензирование проектного решения в процессе разработки программного обеспечения. Однако в большей степени эти семинары напоминали поэтические чтения, что, безусловно, не является типичным. Специальные правила, выработанные на семинарах создателей шаблонов (которые можно рассматривать как работу секций конференций PLoP), оказались мощным средством, способствующим написанию шаблонов программного обеспечения в простой, общедоступной и стандартной форме. Эти правила получили название **шаблонов**, т.е. стандартных приемов на соответствующем уровне абстракции. В работе [661] сообщается об эффективности создания культуры шаблонов в телекоммуникационной компании AGCS. Шаблоны вошли "в моду" в таких компаниях, как IBM, Siemens и AT&T, причем не секрет, что все эти фирмы создают собственные внутренние шаблоны программного обеспечения и открыто публикуют их.

Хотя книга Группы Четырех получила заслуженное признание благодаря рассмотрению профилей шаблонов, для многих она была "палкой о двух концах". Шаблоны GoF образуют каталог самостоятельных шаблонов, причем все они соответствуют примерно одному и тому же уровню абстракции. Такой каталог никогда не будет обладать генеративным свойством, которое приписывает себе язык шаблонов Александера, но, справедливости ради, надо сказать, что Группа Четырех вполне допускает, что подобное свойство не является целью их работы.

---

[5] Членами-основателями являлись Кен Ауер (Ken Auer), Кент Бек (Kent Beck), Гради Буч (Grady Booch), Джим Коплиен (Jim Coplien), Уард Каннингам (Ward Cunningham), Хол Хильдебранд (Hal Hilderbrand) и Ральф Джонсон (Ralph Johnson). Первоначально спонсорами были компания Rational и группа OMG.

Книга Группы Четырех содержит 23 полезных шаблона проектирования, включая следующие, особенно интересные и удобные.

- Facade (фасад). Успешно используется при реализации оболочек объектов: сочетает множество интерфейсов в одном.
- Adapter (адаптер). Также полезен для оболочек: преобразует интерфейсы в форму, понятную для клиентов.
- Proxy (посредник). Используется, главным образом, для поддержки распределенности: создает локального "заместителя" удаленного объекта, что позволяет осуществлять доступ к нему.
- Observer (наблюдатель). Помогает объекту оповещать все зарегистрированные объекты о том, что его состояние изменилось, а также удобен при реализации систем "классной доски".
- Visitor (посетитель) и State (состояние). Эти два шаблона успешно используются при реализации динамической классификации.
- Composite (составной объект). Позволяет клиентам унифицированным образом трактовать части и целое.
- Bridge (мост). Способствует отделению описаний интерфейсов от их реализаций.

Некоторые циники утверждают, что шаблоны GoF на самом деле полезны только для исправления недостатков языка C++. И примерами таких шаблонов якобы могут служить Decorator (декоратор) и Iterator (итератор). Однако само это предположение поднимает вопрос о сопоставлении шаблонов, зависящих от языка и не зависящих от него. В книге [133] (авторы которой работают в компании Siemens в Германии и известны как Партия Пяти, или Party of Five — PoV) предлагается следующая классификация шаблонов: архитектурные, шаблоны проектирования и языковые идиомы. В книге представлены примеры шаблонов первых двух категорий. К архитектурным шаблонам относятся: Pipes And Filters (конвейеры и фильтры), Blackboard (системы классной доски) и шаблон MVC (Model View Controller), предназначенный для разработки пользовательских интерфейсов. Типичные шаблоны проектирования от Партии Пяти (PoV) имеют следующие названия.

- Forwarder-Receiver (отправитель-получатель)
- Whole-Part (целое-часть)
- Proxy (посредник)

Кроме того, Партия Пяти советует читателям использовать все шаблоны GoF. Поэтому их книга может считаться дополнением оригинального каталога, но не только ввиду добавления новых шаблонов, но также и благодаря рассмотрению различных уровней абстракции. Шаблон Whole-Part представляет собой полную реализацию структур композиции, которые составляют часть базовой семантики объектного моделирования. В этом смысле он выглядит как обычный шаблон. Однако поскольку почти все языки не поддерживают подобную конструкцию, может оказаться полезным предоставление стандартного способа реализации такой конструкции. Он представляет собой тот редкий пример шаблона анализа, который

непосредственно соответствует идиоме в различных языках — мультиязыковой идиоме. Наилучшим из известных источников, описывающих идиоматические (т.е. зависящие от языка) шаблоны, является книга Джима Коплиена [191], посвященная современному языку C++; она вышла на три года раньше книги Группы Четырех. К "шаблонам" языка C++ (в книге этот термин не используется), представленным Коплиеном, относятся следующие.

- Handle Class (класс-обработчик) используется для инкапсуляции классов, которые содержат в себе развитую логику приложения.
- Reference Counter (счетчик ссылок) управляет подсчетом ссылок, задействованных в совместно используемом представлении.
- Envelope-Letter (конверт-письмо) позволяет осуществлять "миграцию типов" классов.
- Exemplar (экземпляр) разрешает создание прототипов в случае отсутствия делегирования.
- Ambassador (представитель) обеспечивает прозрачность распределения.

Опытные программисты, работающие в области объектно-ориентированных технологий, сразу же узнают многие из этих шаблонов, и практически любой программист распознает в них идеи, лежащие в основе кэш-памяти и рекурсивных композиций. Кроме того, они могут рассматриваться как шаблоны проектирования и/или анализа. Шаблон Cache (кэш) следует применять в том случае, если при выполнении сложных вычислений удобнее сохранять результаты, чем часто пересчитывать их заново. Этот шаблон также используется тогда, когда затраты на передачу данных по всей сети настолько велики, что эффективнее сохранять их локально. Этот шаблон, безусловно, в значительной степени способствует оптимизации производительности. Следует также отметить, что шаблоны могут использовать друг друга. Этот шаблон может использовать шаблон Observer (см. ниже), если необходимо знать, какие результаты должны быть пересчитаны заново или какие данные должны быть обновлены, что может быть сделано быстро или отложено (в зависимости от относительных параметров чтения и обновления).

Хотя для раздельного документирования таких структур, как наследование или агрегирование, такая практика зачастую является достаточно хорошей (если не учитываются производные зависимости), но, как указывалось в главе 6, шаблоны документирования нередко требуют описания при помощи не одной, а нескольких структур. Хорошими примерами могут служить рекурсивные наращиваемые структуры типа двоичных деревьев и списков: список рекурсивно описывается с помощью атомарной головы и хвоста, который сам по себе является списком, а программа состоит из элементарных команд и блоков, которые в свою очередь тоже состоят из блоков и команд. Для документирования ситуации, представленной на рис. 7.3, можно использовать шаблон метода Catalysis, заменяя обозначение `<Node>` на `Block` (блок), `<Branch>` — на `Program` (программа) и `<Leaf>` — на `Statement` (команда).

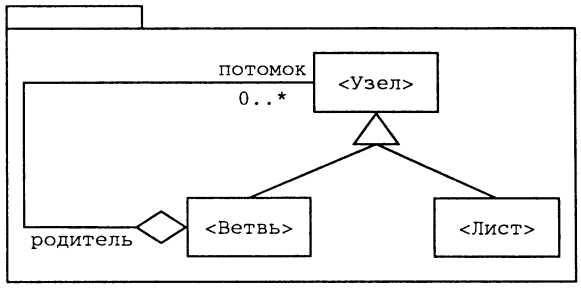


Рис. 7.3. Рекурсивные композиты

Рекурсивные структуры обрабатываются при помощи рекурсивного спуска, поэтому в шаблоне необходимо указать, можно ли предупредить зацикливание посредством включения ограничений. На языке объектных ограничений OCL (Object Constraint Language) такое ограничение может быть записано в следующем виде[6].

```
context Node::ancestors = parent + parent.ancestors AND
context Node:: not(ancestors include self)
```

Шаблоны списков и блоков можно рассматривать как целые объекты или оболочки, и можно считать, что шаблон просто описывает их внутреннюю структуру.

Приведенные выше примеры свидетельствуют о том, что весьма полезно описывать шаблоны в стандартном виде, и многие специалисты используют в качестве стандартной форму, приведенную в работе Александера, — так называемая форма Александера, согласно которой описания шаблонов включают описательные разделы, содержащие соответствующие наглядные иллюстрации следующего вида (хотя на самом деле у разных авторов названия могут отличаться).

- Имя шаблона и описание.
- Контекст (проблема) — ситуации, в которых шаблоны могут быть полезны, и проблема, которую шаблон решает.
- Силы — противодействующие силы, которые должны быть сбалансированы проектировщиком.
- Решение — принципы, лежащие в основе шаблона, и способы его применения (включая примеры их реализации, последствия и преимущества).
- Связанные шаблоны — другие названия для (фактически) тех же шаблонов или шаблоны, в сочетании с которыми может применяться данный.
- Известные случаи использования.

---

[6] Строго говоря, сложение множеств в OCL не определено. Знак плюс — это символ, применяемый в методе Catalysis и позволяющий использовать преимущество симметрии. Правильная форма записи на языке OCL выглядит так: `Set1 -> union(Set2)`.

Кент Бек выпустил книгу, посвященную 92 идиомам языка Smalltalk [62]. Существует также несколько зависящих от языка "версий" книги Группы Четырех, а именно [26] для языка Smalltalk и [190, 336] для Java. Хотя многие из архитектурных шаблонов Партии Пяти также входят в состав "стилей" SEI (института программной инженерии), важно отметить различие их целей. И шаблоны, и стили созданы на основе наилучшей практики разработки программного обеспечения, но стили института программной инженерии предназначены для сбора данных и их формализации (и, по-видимому, последующей автоматизации обработки), а шаблоны Партии Пяти — для дальнейшего обобщения такой практики.

Подавляющее большинство шаблонов программного обеспечения, созданных к настоящему моменту, — это шаблоны проектирования, соответствующие различным уровням абстракции. Однако в [282] в качестве альтернативы шаблонам проектирования приводится идея шаблонов анализа. Шаблоны Фовлера (Fowler) представляют собой многократно используемые фрагменты объектно-ориентированных моделей описания, достаточно общие для того, чтобы применяться в нескольких конкретных предметных областях. Поэтому они несколько напоминают шаблоны GoF (которые представлены в подзаголовке книги как элементы многократно используемого объектно-ориентированного программного обеспечения), но еще более отдалены от концепции Александера. Приведем примеры шаблонов Фовлера.

- Party (участник) — как сохранить имя и адрес, который может понадобиться.
- Organization Structure (организационная структура) — как представить сложную структуру.
- Posting Rules (правила регистрации) — как представлены основные правила учета.

Шаблонов на самом деле намного больше, и некоторые из них предназначены специально для таких областей, как медицина или бухгалтерский учет.

Проблема, связанная с этими шаблонами, заключается в том, что даже самые простые из них (например, шаблон Accountability (средства учета)) действительно очень сложны для понимания по сравнению со сложностью проблемы, которую они решают. Один из специалистов три раза прочитал текст, пока на самом деле не понял суть происходящего. И по завершении этого процесса оказалось, что он уже знал предложенное решение, но никогда не стал бы описывать его подобным образом.

В [511] предложен язык шаблонов, предназначенный для проектирования общественно-технических систем и позволяющий подтвердить правильность требований. Он основывается на модели CREWS-SAVRE, рассмотренной в главе 8. Эти авторы определяют три шаблона следующего вида.

- Machine-Function (машина-функция). Этот шаблон представляет правило, связывающее действие пользователя (сценарий задания на нашем языке) с техническими требованиями к системе с целью поддержки этого действия (операции бизнес-объекта, который реализует это задание). Мы считаем, что это правило можно назвать шаблоном лишь с натяжкой.
- Collect-First-Objective-Last (достижение первой цели в последнюю очередь). Этот шаблон предполагает выполнение пользователем главной транзакции после всех второстепенных. Например, торговые автоматы должны сначала потребовать карточку, а

затем выдать наличные. (Психологическое явление завершения применительно к проектированию пользовательского интерфейса описано в главе 9.)

- Insecure-Secure-Transaction (небезопасная безопасная транзакция). Этот шаблон предполагает, что системы должны контролировать свое состояние безопасности и предпринимать соответствующие действия, если система становится ненадежной.

Ценность этих шаблонов может быть подвергнута сомнению, поскольку, так же как и аналитические шаблоны Фовлера, они, как оказалось, утверждают очевидное. Они не позволяют решить ряд задач, связанных с уточнением прецедентов. Кроме того, можно показать, что они являются ничем иным, как принципами проектирования; точно так же как принцип *завершенности* является известным приемом проектирования интерфейса HCI (host computer interface). Но, с другой стороны, операционализация (operationalization) в системе CREWS-SARVE указывает на то, что они могут иметь практическое значение в определенных специальных случаях в рамках этого и некоторых других контекстов.

Другие типы шаблонов

В последние годы представляет интерес создание шаблонов организации разработки [192, 598–600]. Коплиен применяет идею шаблонов непосредственно для самого процесса разработки программного обеспечения и отмечает некоторые заслуживающие внимания закономерности. Эти наблюдения появились в результате исследовательского (финансируемого компанией AT&T) проекта, посвященного исследованию значимости стандартов, относящихся к процессу контроля качества, например ISO9001, и модели развития функциональных возможностей (Capability Maturity Model) института программной инженерии (SEI). Опираясь на базовый принцип, гласящий, что реальные процессы можно охарактеризовать с помощью их коммуникационных магистралей, Коплиен вместе с Брэдом Кэйном (Brad Cain) и Нейлом Харрисоном (Neil Harrison) проанализировали более 50 проектов, выполненных высокопроизводительными организациями средних размеров, занимающимися разработкой программного обеспечения. Среди этих фирм была компания Borland, разрабатывающая программный продукт электронных табличных расчетов Quattro Pro. Используемая технология заключалась в применении карточек CRC (Class, Responsibility and Collaboration) и выяснении (на семинарах) всех ролей с помощью представителей организаций (в противоположность описаниям заданий). Затем идентифицировались и систематизировались отношения между этими ролями (степень взаимосвязи характеризовалась как слабая, средняя или сильная) с последующим "проигрыванием" процесса разработки, чтобы убедиться в правильности принятых решений. После этого информация вводилась в написанную на языке Smalltalk систему Pasteur, которая выдавала самые разнообразные социометрические диаграммы и метрики. На основании этих исследований были выявлены типичные ключевые характеристики наиболее производительных организаций и был разработан язык шаблонов, состоящий из 42 элементов, описывающий построение организаций-разработчиков. Этот язык содержал шаблоны следующего вида.

- Conway's Law (закон Конвэя) утверждает, что архитектура всегда соответствует организации или наоборот.
- Architect Also Implements (архитектор тоже участвует в реализации) требует, чтобы архитектор не слишком отдалялся от процесса разработки.

- Developer Controls Process (процессом управляет разработчик) —разработчики должны целиком владеть и управлять процессом разработки, а не просто его реализовать.
- Mercenary Analyst (наемный аналитик) обеспечивает обратное проектирование и разрабатывает проектную документацию.
- Firewall (брандмауэр) описывает способ защиты разработчиков от "белого" шума.
- Gatekeeper (привратник) описывает, как разработчики программного обеспечения могут регулярно получать полезную информацию.

Типичным случаем использования подобных организационных шаблонов является совместное применение шаблонов Gatekeeper и Firewall, например, при использовании в пилотном проекте новой технологии. Отрасль разработки программного обеспечения всегда была подвержена распространению слухов, ситуация усугублялась привычками поставщиков, которые делали хвастливые заявления задолго до появления конкурентоспособных реализаций. Излишнее внимание к ложным слухам в этой отрасли, не говоря уже о кажущихся авторитетными утверждениях в профессиональной прессе, может серьезно подорвать доверие к пилотному проекту. Например, разработчики теряют доверие к языку Java, поскольку он характеризуется низкой производительностью или отсутствием доступных инструментальных средств. Наряду с этим некоторые новости имеют весьма важное значение — например, появление версии 2 языка Java. Хорошие новости разработчики должны знать. Поэтому необходимо построить официальные брандмауэры, а затем ввести роль привратника. Тогда назначенный представитель или, возможно, виртуальный центр информации об объектах будет нести ответственность за отбор и продвижение полезной и пригодной к использованию информации (в противоположность безосновательным выдуманным историям, рекламным почтовым рассылкам и даже "заботам" продавцов и поставщиков).

Однако гораздо больший интерес представляют не отдельные шаблоны, а сам подход к языку Коплиена, который по своему духу намного ближе к работе Александера, чем к любой другой информации, изложенной в книгах Группы Четырех или Партии Пяти. Во-первых, поскольку этот язык описывает взаимодействие между людьми, основное внимание в нем сконцентрировано на человеке. Во-вторых, его цели представляются явно генеративными. Коплиен утверждает, что хотя разработчики программного обеспечения не "населяют" код подобно тому, как люди заполняют дома и офисы, но как профессионалы они могут выступать в роли экспертов профессиональных процессов и организаций. Поэтому, ориентируясь на язык Александера, который предусматривает вовлечение в процесс разработки всех заинтересованных лиц (и, в первую очередь, пользователей зданий), создатели формального процесса тоже должны основываться на опыте "жертв" — самих разработчиков. Попытка Коплиена обновить язык шаблонов Александера, как оказалось, увела эти шаблоны в сторону от описания фрагментов структуры (что типично для шаблонов GoF), но в значительной мере приблизила их к описанию работы, которая должна быть выполнена. Если "уйти" от чистой структуры, шаблоны Коплиена выглядят истинно архитектурными в большей степени, чем существующие шаблоны многих других типов.

На самом деле очевидно, что из общих корней произрастают два полярных представления шаблонов, получившие широкое распространение в настоящее время. Одно представление рассматривает шаблоны как общие структурные описания. Они были описаны (особенно в книгах по UML) как "параметризованные виды сотрудничества". Идея, лежащая в их основе, такова: можно взять, к примеру, структурные описания ролей, которые различные классы

могут играть в шаблоне, и затем, просто изменив имена классов и предоставив подробную реализацию алгоритмов, "встроить" их в разрабатываемую программу. Шаблоны, таким образом, превращаются в абстрактные описания потенциально подключаемых компонентов. Проблема, связанная с подобным упрощенным представлением, возникает в том случае, когда требуется класс, исполняющий в разных шаблонах несколько ролей одновременно. Например, недавно Эрик Гамма заново реализовал каркас HotDraw на языке Java. В нем один класс — `Figure` — может сотрудничать с четырнадцатью различными взаимосвязанными шаблонами. Сложно представить успех подобного проектного решения, если бы каждый из таких шаблонов инстанцировался как отдельный компонент. И что еще более важно, такое представление ничего не говорит о том, каким образом следует объединять разные проектные решения. Оно лишь утверждает, что они (их фрагменты) должны быть похожими с точки зрения структуры. В другом случае шаблоны просто считаются проектными решениями (рассматриваемыми в определенном контексте типичной проблемы). Такое представление неминуемо приводит к разработке шаблонов как элементов генеративного языка.

Этой точки зрения придерживается в своей работе Алан О'Каллаган (Alan O'Callaghan) и его коллеги из группы объектной инженерии, лаборатории, занимающейся исследованиями в области технологий программного обеспечения (Software Technology Research Laboratory) и университета Де Монтфорта (De Montfort). О'Каллаган является главным автором языка шаблонов ADAPTOR, предназначенного для перевода уже существующих систем в объектную и компонентную структуры. Первоначально язык ADAPTOR основывался на опыте пяти проектов, выполняемых начиная с 1993 году в различных предметных областях. Для модернизации объектной структуры использовался подход, основанный на архитектуре и шаблонах. В настоящее время в нем интегрирован опыт, полученный в восьми серьезных проектах, выполненных в четырех различных областях: телекоммуникации, розничной торговли, обороны и разведки месторождений нефти. О'Каллаган утверждает, что переход к объектной технологии — это не просто восстановление структурной схемы и алгоритма работы по исходным текстам, поскольку такое обратное проектирование обычно является формальным, сконцентрировано исключительно на функциональных возможностях существующих систем и предполагает создание архитектуры, которая похожа на саму себя в оригинале. Самая важная информация, касающаяся логического обоснования исходного проектного решения, уже безвозвратно утеряна. Она не может быть извлечена из кода, поскольку код никогда не содержал эту информацию (если он, конечно, не был написан в необычной для него выразительной манере). Самое лучшее, чего могут добиться традиционные археологические методики обратного проектирования, — это воссоздать прежнюю систему в объектно-ориентированном "стиле", который чаще всего не предоставляет ни одного из необходимых преимуществ.

Подход, впервые примененный Грэхемом [327] и О'Каллаганом, состоит в разработке объектных моделей требуемой "новой" и существующей систем, а затем с помощью специалистов по сопровождению и разработчиков (а не с использованием кода или проектной документации) определяется, какие элементы системы могут использоваться в прежнем виде, а какие необходимо реализовать заново. Группа О'Каллагана обратилась к шаблонам в поисках некоторого способа документирования и объединения обычных практических приемов, которые оказались успешными в каждом новом проекте (существующие системы доставляют особенно много проблем и являются, в общем и целом, всегда уникальными по отношению к самим себе). Вначале использовались готовые шаблоны проектирования, относящиеся к данной области, но затем команда сосредоточила усилия на разработке собственных шаблонов. Далее проблемы, связанные с правом собственности на код (т.е. ответственности за ту

часть модифицируемой системы, которая принадлежит кому-то другому, а не непосредственно самому клиенту) и вызванные тем, что переход обычно предполагает радикальные изменения на уровне макроструктуры системы, потребовали создания шаблонов и для решения организационных проблем и проблем, связанных с процессом разработки. И наконец, исследования показали, что самые мощные шаблоны, предназначенные для использования в различных областях, оказались взаимосвязанными, ввиду чего появилась возможность создания генеративного языка шаблонов.

Язык ADAPTOR в 1998 году был объявлен "подходящим претендентом на роль открытого, генеративного языка шаблонов". Он рассматривался в качестве языка-кандидата по двум причинам. Во-первых, несмотря на огромный успех проектов, положенных в его основу, ADAPTOR не был в достаточной степени всеобъемлющим с точки зрения области сферы действия или рекурсивным по отношению к достаточному уровню детализации, т.е. действительно генеративным. Во-вторых, О'Каллаган доверял различным шаблонам в разной мере. Он считал совершенными только те шаблоны, которые прошли обсуждение на семинарах. Шаблоны, не утвержденные на семинарах, автор рассматривает в качестве кандидатов. Язык ADAPTOR можно считать открытым по ряду причин. Во-первых, как и в любом настоящем языке, и сам по себе язык, и входящие в его состав элементы допускают возможность развития. Многие из наиболее развитых шаблонов, например Get The Model From The People (получить модель у специалистов), представленный впервые в 1996 году на семинаре TelePlop, выдержали несколько циклов изменений. Во-вторых, придерживаясь взглядов авторов работы [23], О'Каллаган настаивает на том, что шаблоны сами по себе являются открытыми абстракциями. Так как ни один настоящий шаблон не предоставляет исчерпывающего решения и при каждом его применении получаются разные результаты (поскольку он используется в различных контекстах), он не подвержен формализации, которой могут быть подвержены истинные абстракции, например правила. Наконец (и этот случай уникален для всех опубликованных языков шаблонов, относящихся к программному обеспечению) ADAPTOR является открытым, поскольку он явным образом использует другие языки шаблонов и каталоги, например уже упоминавшийся генеративный язык шаблонов процесса разработки, созданный Коплиеном, или каталоги GoF или PoV.

Язык ADAPTOR содержит следующие шаблоны.

- Get The Model From The People (получить модель у специалистов) предполагает использование специалистов по сопровождению существующих систем в качестве источников информации о бизнес-процессах.
- Pay Attention To The Folklore (учитывать народную мудрость) предлагает рассматривать группы разработки и сопровождения в качестве экспертов по данной области, даже если они не считают себя таковыми.
- Buffer The System With Scenarios (проигрывать в системе резервные сценарии) предполагает привлечение главных бизнес-аналитиков, маркетологов, футурологов и других специалистов для проигрывания игровых бизнес-сценариев, напрямую не относящихся к их компетенции в рамках спецификации требований.
- Shamrock (трилистник) разделяет систему на три тесно взаимосвязанных "листа", причем каждый из них может содержать множество классов различных категорий и пакетов. Листья представляют собой концептуальную область (объекты пространства задачи), область инфраструктуры (перманентность, параллельность и т.д.) и область

взаимодействия (графические пользовательские интерфейсы, межсистемные протоколы и т.д.).

- Time-Ordered Coupling (упорядоченное по времени связывание) предполагает разбиение классов на кластеры в соответствии с общепринятыми коэффициентами изменчивости, чтобы обеспечить гибкость к изменениям.
- Keeper Of The Flame (хранитель огня) определяет роль, обеспечивающую согласованность деталей проектного решения с общей архитектурой: изменения макроструктуры допустимы только в том случае, если они являются необходимыми и уместными.
- Archetype (архетип) предполагает создание типов объектов, представляющих ключевые абстракции, выявленные в пространстве задачи.
- Semantic wrapper (семантическая оболочка) предполагает создание оболочек для существующего кода, которые представляют поведенческие интерфейсы абстракций, распознаваемые остальной частью системы.

Открытый и генеративный характер языка ADAPTOR подтверждается при использовании шаблонов на ранних стадиях проектирования, связанного с переводом существующих систем на новые технологии. В основе языка шаблонов ADAPTOR лежит описанный ранее подход, основанный на построении модели. Модели пространства задачи О'Каллагана включают в себя типы объектов и существующие между ними отношения, которые описывают поведение не только ключевых абстракций в рамках контекста системы, но и самой системы. Поэтому Archetype является одним из первых шаблонов, применяемых совместно с шаблонами Get The Model From People и Pay Attention To The Folklore. Применяемые на ранних стадиях стратегические сценарии "что..., если..." не соответствуют этой модели, но укладываются в рамки шаблона Buffer The System With Scenarios. Шаблон Shamrock применяется для отделения типов объектов предметной области от чисто системных ресурсов, которые должны предоставляться на стадии выполнения. "Лист" предметной области затем может быть включен в другие пакеты при помощи шаблона Time-Ordered Coupling с целью совместной поддержки типов с одинаковой степенью изменчивости (выявленных в ходе проигрывания сценариев). Шаблон Коплиена Conway's Law применяется для планирования организации разработки, которая должна соответствовать развертывающейся структуре системы. Еще один шаблон Коплиена Code Ownership (право собственности на код) гарантирует, что каждому пакету присваиваются свои обязанности. Шаблон языка ADAPTOR, получивший название Trackable Component, гарантирует, что такие "владельцы кода" несут ответственность за публикацию тех интерфейсов пакетов, которые должны разрабатываться совместно с другими, чтобы их развитие было управляемым. Шаблон GoF Facade применяется для создания "внешнего обрамления", скрывающего детальную структуру системы. Именно на этом этапе принимается решение о том, какие части функциональности требуют нового кода, а в каких можно использовать существующий. "Внешнее обрамление" гарантирует, что подобные решения (а также результаты их реализации) находятся под полным контролем и не вызовут непредусмотренных волновых эффектов. Если используется существующий код, то для связывания прежнего, уже существующего продукта с новыми объектно-ориентированными фрагментами применяется шаблон Semantic Wrapper.

Даже из такого поверхностного примера можно видеть, как этот язык решает все важные проблемы архитектуры (вопросы требований клиентов, концептуальной целостности, структуры, процесса, организации и т.д.) и "принимает близко к сердцу" проблемы перехода

существующих систем. О'Каллаган рассказывает о том, что во время общего представления этого подхода на открытой лекции один слушатель высказал свои возражения и заявил, что подход на основе построения модели нельзя считать "перепроектированием" — это просто "прямое проектирование", предусматривающее многократное использование некоторой части существующего кода". В ответ О'Каллаган выразил свое согласие и подтвердил, что это именно так и есть. В ходе дальнейшего исследования оказалось, что многие из шаблонов языка ADAPTOR вообще не были предназначены для перевода существующих систем на новые технологии. В результате в настоящее время язык ADAPTOR рассматривается как подмножество более общего языка, предназначенного для практического архитектурного строительства в области разработки программного обеспечения, под кодовым названием проектирования Януса [603].

Дебаты относительно природы шаблонов программного обеспечения ("параметризованные виды сотрудничества" или "проектные решения"; каталоги или языки шаблонов) уже сами по себе отражают дискуссии по поводу архитектуры программного обеспечения, рассмотренной в предыдущем разделе. Эта взаимосвязь была полностью раскрыта после назначения Коплиена на должность приглашенного редактора журнала по программному обеспечению *IEEE Software* осенью 1999 года. Выпуск был специально посвящен архитектуре программного обеспечения, и Коплиен, помимо всего прочего, опубликовал основные идеи доклада Александера на конференции OOPSLA, прошедшей в 1996 году в г. Сан Жозе, Калифорния [25]. В своей передовой статье, по-новому оценивающей модельное представление архитектуры, Коплиен обозначил два основных подхода к разработке программного обеспечения: методология предварительного проектирования ("кальки") или "генерального планирования", которая противопоставлялась технологии "постепенного наращивания" [193]. Коплиен высказал предположение, что неразвитая дисциплина архитектуры программного обеспечения страдает от "формальной зависти" и перенимает неподходящие уроки как из мира разработки аппаратных средств, так и из строительной среды. К симптомам ее кризиса можно отнести отделение компонентов архитектуры от артефактов, поставляемых потребителю, и рассмотрение архитектуры в качестве отдельного процесса в рамках каскадного метода разработки программного обеспечения. Поддерживая мнение архитектора Людвига Майлса ван дер Роха (Ludwig Miles van der Rohe), работающего в области строительства, Коплиен, как и Александер, заявляет, что "Бог обитает в деталях" и что ясность на макроуровне достигается только за счет успешного связывания мелких деталей. Далее он утверждает, что "опыт работы с объектами показал то, что было важным всегда: качество архитектуры определяется не столько программным обеспечением, сколько людьми, которые его пишут" (с. 41).

Основные идеи связывания и сцепления позволяют людям работать над созданием некоторой части программного обеспечения и не только понимать, но и оценивать свой конкретный вклад. Коплиен считает карточки CRC и их использование в объектно-ориентированной разработке классическим примером, свидетельствующим об антропоморфной природе проектирования программного обеспечения. С этой точки зрения шаблоны программного обеспечения явились следующей волной, предваряющей деятельность такого рода в области программной архитектуры. Как вполне справедливо указывает Коплиен, развитие шаблонов для слабого программиста всегда служило основным источником знаний об архитектуре при разработке программного обеспечения. Кроме того, архитектура признает существование глубокой взаимосвязи между структурой кода и способами связи между специалистами, разрабатывающими и поддерживающими ее. Далее Коплиен приводит доводы в защиту того, что шаблоны начали использоваться в разработке программного обеспечения намного позже той наивной практики первых дней применения объектов, которая резко снижала их возможности,

поскольку все еще была ограничена рамками модельного представления программ, унаследованного от прежней культуры. Дальнейшее продвижение вперед требует освобождения от груза "исторических иллюзий, связанных с формализмом и планированием" (с. 42).

Ричард Габриэль (Richard Gabriel), который является в настоящее время членом группы Hillside Group, квалифицированным специалистом-практиком в области программного обеспечения, а также профессиональным поэтом[7], полагает, что существуют две причины, почему все успешные разработки программного обеспечения должны в действительности придерживаться технологии постепенного наращивания. Во-первых, имеется определенная сложность, связанная с познанием, если рассматриваются не только текущие, но и возможные в будущем случаи изменений. При этом невозможно заранее вполне отчетливо представить себе создаваемую программу на необходимом уровне детализации с произвольной точностью [287]. Во-вторых, известен тот факт, что предварительное планирование отпугивает всех, кроме самих планировщиков. Коплиен, Габриэль и все приверженцы шаблонов целиком придерживаются такой практики разработки, которая борется с подобным общественным отчуждением. При этом они придают огромное социальное и моральное значение понятию архитектуры программного обеспечения. Перед лицом этой суровой реальности Дэвидом Парнасом (David Parnas) был предложен один вариант, который можно рассматривать в качестве альтернативы постепенному наращиванию: как только код будет завершен, можно "подделать кальки" посредством обратного проектирования системы по исходным текстам.

7.2.1. Шаблоны проектирования для разделения

Для того чтобы на самом деле получить преимущества от полиморфизма (или "подключаемости"), в программе важно объявлять переменные и параметры не в явных классах, а через интерфейс (абстрактный класс в языке C++, интерфейс в языке Java, отложенный (абстрактный) класс в языке Eiffel). Рассмотрим модель кафе, изображенную на рис. 7.4. Исходная модель содержит класс `Food` (пища), который используется двумя классами `Kitchen` (кухня) и `Cash Register` (кассовый аппарат). Но эти клиенты требуют от класса `Food` различного поведения, поэтому выделим их требования в разные интерфейсы: `Edible` (съедобная) — для кухни, нуждающейся в запасах еды, и `Saleable` (продаваемая) — для кассы. Поступая так, мы сделали проектное решение более гибким, так как теперь можно рассматривать съедобные объекты, которые не являются продаваемыми (ингредиенты, например мука), и продаваемые, которые не являются съедобными: теперь можно начать продавать в кафе газеты. Исходному классу `Food` приходится реализовывать оба интерфейса. Некоторые хорошие программисты настаивают на том, что при объявлении переменных и параметров всегда нужно использовать интерфейсы. Простое множественное наследование типов-ролей может рассматриваться как шаблон, предназначенный для формирования видов сотрудничества. (В тех языках, где не предусмотрено использование типов, например в языке Smalltalk, различие проявляется только в модели проектирования, но не в программе.)

Использование интерфейсов — это основной шаблон, позволяющий уменьшить зависимости между классами. Класс представляет реализацию; интерфейс задает описание того, что требует определенный клиент. Таким образом, объявление интерфейса снижает до минимума зависимость клиента от других классов: все, что нужно, — это соответствовать описанию, предоставленному интерфейсом.

---

[7] Правила проведения семинаров авторов шаблонов, определяющие стиль работы конференций PLoP, приписываются Габриэлю.

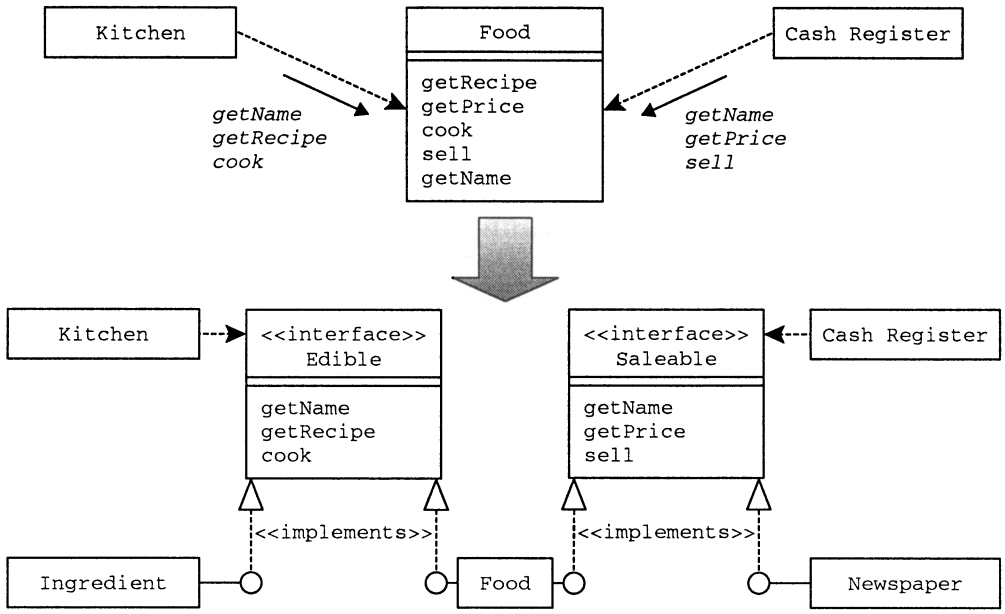


Рис. 7.4. Разделение при помощи интерфейсов

Разделение с использованием фабрики

Если в программу добавляется новый класс, необходимо как можно меньше изменять остальной код программы. Поэтому следует свести к минимуму количество явных упоминаний классов. Как уже отмечалось, это можно сделать с помощью объявления переменных и параметров — вместо имен классов использовать имена интерфейсов. К другим точкам вариаций относятся те фрагменты программы, где создаются новые экземпляры. В некотором месте программы должен быть осуществлен явный выбор класса (при помощи команды `new Classname` или в некоторых языках — посредством клонирования существующего объекта).

Шаблон Factory (фабрика) используется для уменьшения зависимости такого рода. Все случаи создания сконцентрированы в одном месте — **фабрике** (factory). Фабрика обязана "знать", какие классы и где находятся, а также какой класс должен быть выбран в конкретном случае. Фабрика может представлять собой метод или объект (или, возможно, одну из ролей объекта с соответствующими обязанностями). В качестве примера пользователь может создать в графическом редакторе новые фигуры, выбирая на клавиатуре клавишу <c> для создания нового кружка, <r> — для нового прямоугольника и т.д. Затем от комбинаций клавиш следует перейти к классам, возможно, используя для этой цели оператор `switch` или табличные значения. Все это можно сделать при помощи фабрики фигур, которая обычно имеет метод вроде `makeShapeFor (char keystroke)`. Тогда, чтобы изменить проектное решение и использовать новую фигуру, необходимо добавить новый класс (который будет подклассом класса `Shapes`) и изменить фабрику. Чаще всего в данном случае используется меню, в котором содержатся обозначения из таблицы или названия типов фигур.

Как правило, для каждой изменчивой характеристики проектного решения требуется отдельная фабрика: создания фигур, драйверов указывающих устройств и т.д.

Каждый класс фабрики может иметь подклассы. Различные подклассы могут осуществлять разные ответные действия, в зависимости от того, какие классы должны создаваться. Предположим, что пользователь редактора рисунков может переключаться из режима создания простых фигур в режим декорированных форм. Мы добавляем новые классы (нечто вроде `FancyTriangles`), которые должны использоваться вместе с классами простых фигур, однако, вместо задания других клавиш для их создания, мы задаем переключатель режимов, в задачу которого входит изменение этого указателя в редакторе.

```
ShapeFactory shapeFactory;
```

Обычно он указывает на экземпляр класса `PlainShapeFactory`, который реализует интерфейс `ShapeFactory`. При нажатии клавиши <c> данная фабрика создает стандартный кружок. Но мы можем перенаправить указатель на экземпляр класса `FancyShape Factory`, который при тех же входных данных создает объект `FancyCircle` (нестандартный кружок). В [291] такие классы (подобные классу `ShapeFactory`) называются абстрактными фабриками (abstract factory). Все сообщения фабрик реализуются с помощью метода `makeShapeFor (keystroke)`, однако его подклассы создают различные версии объектов.

Разделение при помощи делегирования

Программисты, которые ранее не занимались объектно-ориентированным программированием, могут чрезмерно увлечься наследованием. Наивные аналитики системы отелей могут прийти к заключению, что существует несколько видов отелей, в которых выделение комнат гостям производится по-разному; а наивный проектировщик сможет на этом основании создать в исходном тексте программы соответствующий набор подклассов класса `Hotel`, перекрывая в различных подклассах метод выделения комнат.

```
class Hotel {...
   public void checkInGuest (...)
   ...
   abstract protected Room allocateRoom (...);
   ...}
class LeastUsedRoomAllocatingHotel extends Hotel
{  protected Room allocateRoom (...)
   {     //выделить наиболее редко используемую комнату
    ...}
}
class EventlySpacedRoomAllocatingHotel extends Hotel
{  protected Room allocateRoom  (...)
   {  //выделить самую дальнюю комнату
```

Подобная тактика не может считаться удовлетворительной, поскольку она не может быть повторена в случае других вариантов требований, например, если имеется несколько способов оплаты труда сотрудников. Перекрытие методов — это один из "тупиков" в истории объектно-ориентированного программирования. На практике этот способ удобен только в рамках контекста немногих конкретных шаблонов (обычно тех, которые связаны с предоставлением поведения по умолчанию). Между тем вся хитрость состоит в том, чтобы перенести каждый отдельный вариант поведения в отдельный объект. Для каждой операции создается

свой интерфейс: для распределения комнат, для выдачи заработной платы сотрудникам и т.д., а затем для них определяются различные конкретные реализации. Можно написать, к примеру, следующее.

```
class Hotel  {
   Allocator allocator; ...
   public void checkInGuest  (...)
   {... allocator.doAllocation (...);..}
...}
interface Allocator{
   Room doAllocation(...); //возвращает свободную комнату
...}
class LeastUsedAllocator implements Allocator
{  Room doAllocation (...) {...код ...}}
class EventSpaceAllocator implements Allocator
{  Room doAllocation (...) {...код ...}}
```

Этот шаблон, позволяющий переместить поведение в другой объект, называется Delegation (делегирование). Он имеет несколько вариантов, которые описываются по-разному в зависимости от цели использования данного варианта. Одним из преимуществ шаблона делегирования является то, что он позволяет заменить объект-распределитель комнат, например, на стадии выполнения посредством "включения" новой реализации `Allocator` (распределитель). Если это приходится делать часто, используется шаблон State (состояние).

Еще один случай применения шаблона Delegation получил название Policy (политика): он предполагает отделение процедур, зависящих от бизнес-процессов, от основного кода, чтобы облегчить их изменение. Распределение комнат может служить примером реализации такого шаблона. Шаблон Policy другого вида после каждой операции над объектом предполагает проверку, позволяющую убедиться в соответствии ограничениям бизнес-процесса. В противном случае генерируется исключительная ситуация и операция отменяется. Например, администратор отеля, расположенного в слаборазвитом районе мира, может захотеть, чтобы молодые люди противоположного пола никогда не занимали соседние комнаты. Соответствие этому правилу необходимо проверять при каждой операции выделения комнаты.

Разделение при помощи событий, наблюдателей и MVC

Интерфейсы устраняют для классов необходимость явного знания реализации других объектов, но все же небольшая информация о действиях другого объекта сохраняется. Например, интерфейс `RoomAllocator` содержит метод `allocateRoom(guest)`. Отсюда понятно, чего ожидает класс `Hotel` от любой реализации интерфейса `RoomAllocator`. Но иногда целесообразно усилить разделение с тем, чтобы отправитель сообщения даже не знал, с чем связано это сообщение. Например, объект отеля мог бы отправлять сообщение заинтересованным сторонам каждый раз, когда комната становится занятой или освобождается. Для работы с этой информацией можно придумать самые разные классы: флажок, который сообщает о текущем состоянии комнаты (занята или свободна), систему резервирования, объект, который направляет уборщиков, и т.д.

Подобные сообщения называются **событиями**. Событие передает информацию, и, в отличие от обычного понятия операции, отправитель не имеет определенных предположений насчет того, какие действия оно вызывает. Он просто передает его получателю. Отправитель

события в состоянии отправить его другим сторонам, которые зарегистрировали свои интересы, но отправитель ничего не должен знать о назначении этих событий. События представляют собой весьма общий пример разделения.

Чтобы иметь возможность отправлять события, объект должен предоставить операцию, при помощи которой клиент может регистрировать свою заинтересованность; и кроме того, объект должен поддерживать список заинтересованных сторон. Всякий раз, когда происходит соответствующее событие, он отправляет стандартное уведомляющее сообщение каждому указанному в списке участнику.

Расширением шаблона Event (событие) является шаблон Observer (наблюдатель). В этом случае отправитель и получатель называются Subject (субъект) и Observer (наблюдатель), и событие отправляется каждый раз, когда происходит изменение состояния отправителя. Поэтому наблюдатели всегда в курсе изменений субъекта. Новые наблюдатели можно без труда добавить как подтипы, что и демонстрирует рис. 7.5.

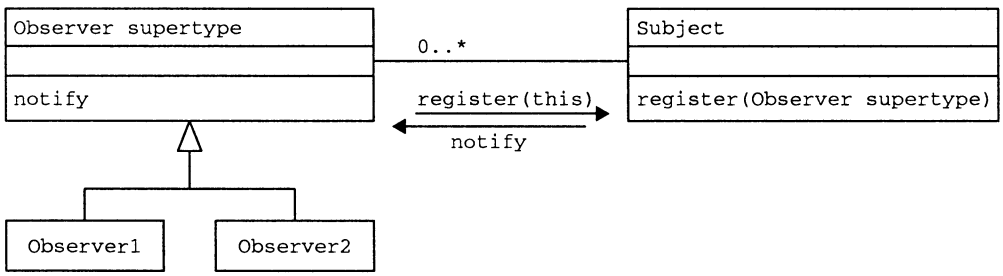


Рис. 7.5. Шаблон Observer

Самый общий случай применения шаблона Observer можно обнаружить в пользовательских интерфейсах: отображаемая на экране информация соответствует текущему состоянию базовых бизнес-объектов. К двум огромным преимуществам такого применения шаблона можно отнести следующие.

1. Пользовательский интерфейс может быть легко изменен без воздействия на бизнес-логику.
2. Несколько представлений некоторого бизнес-объекта могут существовать одновременно, поскольку возможно использование различных классов представления.

Например, проект автомобиля может быть представлен и как технический чертеж, и как счет на материалы: любые изменения, сделанные в одном представлении, немедленно отражаются в другом. Пользователи текстовых процессоров и операционных систем тоже знакомы с подобным механизмом: изменения, внесенные в одном месте (например, изменение имени файла), появляются в другом. И только в очень старых системах для отображения изменения, внесенного в одном месте системы, требуется явный ответ на запрос "вручную". Еще одним повсеместным случаем использования шаблона Observer является его применение в системах, основанных на методологии "классной доски".

Шаблон Observer происходит от шаблона (или "парадигмы", как его часто называют по ошибке) Model-View-Controller, или MVC (модель-представление-контроллер), впервые описанного в 1970 году в работе Тригва Ринскауга (Trygve Reenskaug), посвященной языку

Smalltalk. Сейчас его реализацию можно найти в библиотеке Java AWT и Swing. Модельное представление шаблона MVC повлияло также на некоторые объектно-ориентированные и основанные на объектах языки визуального программирования, например Delphi и Visual Basic.

Модель шаблона MVC представляет собой объект, относящийся к бизнес-логике программы; ее не нужно смешивать с термином "моделирование". **Представление** — это наблюдатель, в задачу которого входит отображение текущего состояния модели на экране или любом используемом устройстве вывода. Он должен находиться в курсе любых изменений, происшедших на данный момент. Иными словами, он преобразует внутреннее модельное представление в представление на экране, удобное для восприятия человеком. Объекты **контроллера** делают обратное: они преобразуют действия человека (нажатия клавиш или перемещения мыши) в операции, которые могут быть поняты объектом модели. Отметим, что на рис. 7.6 шаблон Observer устанавливает двухстороннюю видимость между моделью и контроллером, а также между моделью и представлением. Контроллеры остаются невидимыми для представлений, хотя фрагмент представления может иногда использоваться как контроллер. Примером могут служить ячейки электронной таблицы.

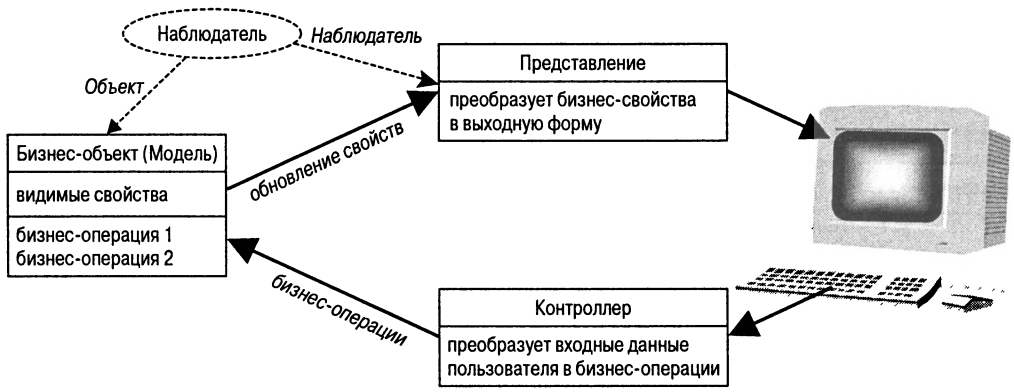


Рис. 7.6. Шаблон Model-View-Controller инстанцирует шаблон Observer

Представления часто бывают вложенными, поскольку они представляют объекты сложной модели, которые вместе с другими объектами образуют иерархии "целое-часть". Каждый класс контроллера обычно используется вместе с определенным классом представления, поскольку интерпретация жестов и нажатий клавиш, производимых пользователем, как правило, зависит от того, на какое представление указывает мышь.

Разделение, выполняемое при помощи адаптера

В шаблоне MVC представление и контроллер являются специализациями шаблона Adapter (адаптер) (не надо путать с языком шаблонов ADAPTOR). **Адаптер** представляет собой объект, связывающий два класса, которые ничего не знают друг о друге, и преобразующий события, выдаваемые одним классом, в операции над другим классом. Представление преобразует события, связанные с изменением свойств модели объекта, в графические операции соответствующей системы управления окнами. Контроллер (Controller) преобразует входные события, связанные с действиями пользователя и нажатиями клавиш, в операции модели.

Адаптер имеет сведения о двух объектах, участвующих в преобразовании. Его преимущество состоит в том, что ни один из объектов не должен иметь информацию о другом объекте, что можно видеть из диаграммы зависимости пакетов, представленной на рис. 7.7.

В общем случае в качестве адаптера может рассматриваться любое семейство функций, но это понятие обычно используется в контексте событий, т.е. там, где отправитель сообщения действительно ничего не знает, что произойдет с этим сообщением.

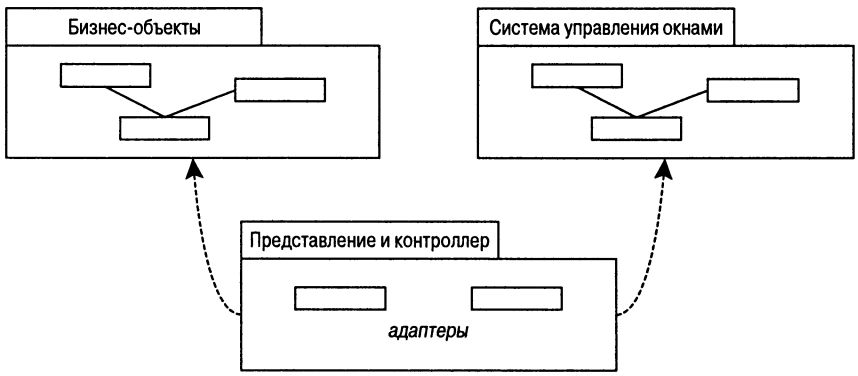


Рис. 7.7. Адаптеры разделяют связываемые ими объекты

Адаптеры применяются не только в пользовательских интерфейсах, но и в самых разнообразных контекстах, в том числе и в крупномасштабных системах: они могут соединять компоненты, работающие в разных сегментах памяти. Адаптеры удобно использовать в качестве связующего звена для двух или нескольких уже существующих или отдельно спроектированных фрагментов программного обеспечения.

Разделение при помощи портов и соединителей

Адаптеры обычно осуществляют преобразование между двумя определенными типами — например, преобразование нажатий клавиш и щелчков кнопкой мыши в графическом пользовательском интерфейсе в операции бизнес-объектов. Это, безусловно, означает, что для связывания различных пар классов всякий раз необходимо проектировать новый адаптер. Зачастую это целесообразно — разные бизнес-объекты требуют различных пользовательских интерфейсов. Но есть еще одна хорошая возможность переконфигурирования набора компонентов без необходимости создания нового набора адаптеров.

На рис. 7.8 показан пример реконфигурируемой системы. Здесь указаны разъемы, связывающие простую электронную систему в одно целое. При этом используется понятие экземпляра/порта или **капсулы** (capsule) из UML, предназначенное для проектирования систем реального времени. Порты связываются при помощи **соединителей** (connector). Соединитель не обязательно реализуется как фрагмент программного обеспечения, обладающий собственными правами: зачастую это просто протокол, о котором условились проектировщики портов. Мы пытаемся свести к минимуму количество типов соединителей, чтобы увеличить до максимума возможности любой пары связанных портов. В нашем примере имеются соединители двух типов, которые можно назвать **соединителями событий** (event connector) и **соединителями свойств** (property connector); они передают соответственно простые события и

вычисленные атрибуты. Вы можете рассматривать эти компоненты как физические устройства или программное обеспечение. Интерфейс кнопки в случае ее нажатия всегда передает одно и то же напряжение (или сигнал). Интерфейсы пуска и останова интерпретируют этот сигнал по-разному в соответствии с реализацией конечного автомата двигателя. Относительно такого набора компонентов можно сказать, что тщательное проектирование протоколов интерфейсов и точек подключений позволяет использовать их для абсолютно (хорошо, не совсем абсолютно!) разных целей, что и демонстрирует рис. 7.9. Детали этого набора могут монтироваться по-разному, в результате чего создаются самые разные конечные продукты. Это напоминает игрушечный конструктор. Секрет возможности реконфигурирования заключается в том, что каждый компонент привносит свой собственный адаптер, который осуществляет преобразование его сигналов в общий "язык", понятный многим компонентам. Такие встроенные адаптеры называются портами. На рисунке они изображены в виде небольших прямоугольников.

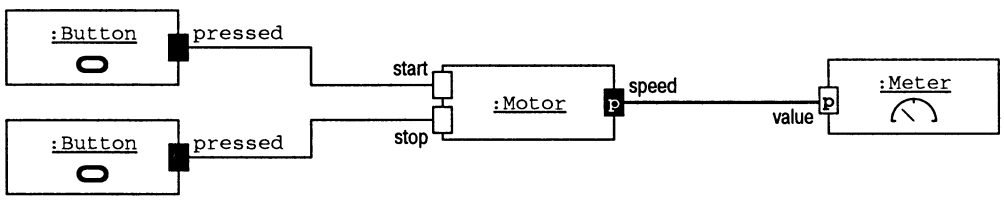


Рис. 7.8. Порты компонентов, а также соединители событий и свойств

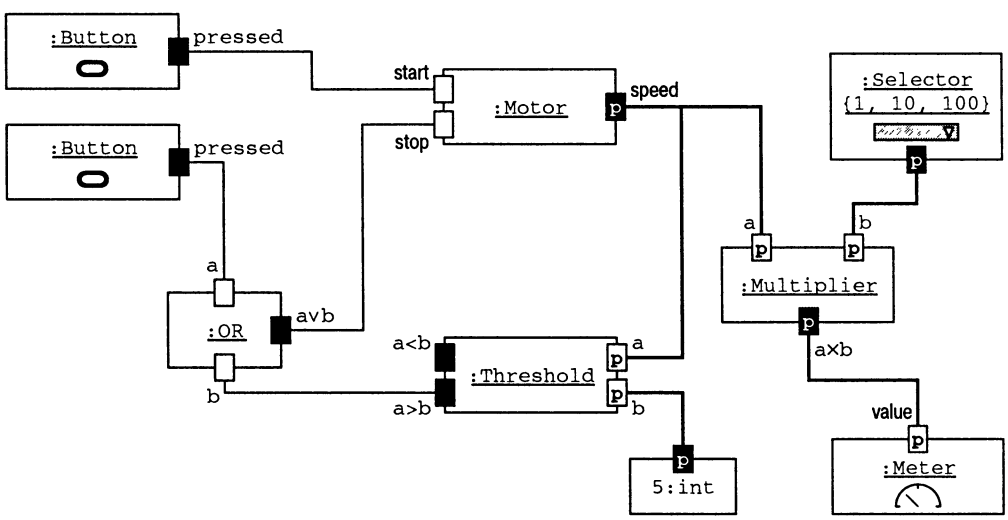


Рис. 7.9. Создание различных продуктов из одних и тех же компонентов

Соединитель представляет собой настолько полезную абстракцию, что заслуживает собственной системы обозначений. Мы использовали расширение языка UML для систем реального времени (UML-RT), допустив при этом некоторую вольность художника с целью выделения двух типов соединителей. В этой системе обозначений объекты помечены стереотипом

<<capsule>>. Небольшие черные прямоугольники в этом примере представляют выходные порты, а белые — входные порты. Символ p указывает на аналоговый порт "свойства": такие порты передают получателю значения при изменении состояния отправителя — иными словами, реализуют шаблон Observer. Непомеченные порты передают дискретные события. Соединители свойств показаны более жирными линиями, чем соединители событий.

Если понятно, что представляют собой порты, то можно идти дальше и проектировать нужные продукты из наборов компонентов, не заботясь обо всех деталях регистрации интересов, оповещающих сообщениях и т.д.

С портами могут быть связаны достаточно сложные протоколы. Каждый компонент должен быть достаточно хорошо определен, чтобы он мог использоваться без необходимости проникновения в его внутреннее устройство. Итак, давайте рассмотрим один из способов работы порта, представленный на рис. 7.10. Выходной порт предоставляет сообщения, позволяющие другому объекту регистрировать интересующие его события (например, когда пользователь касается компонента кнопки). Порт отправляет стандартное "оповещающее" сообщение всем зарегистрированным сторонам. Входной порт реализует интерфейс Listener (приемник), предназначенный для подобных оповещающих сообщений: при получении такого сообщения он отправляет соответствующее сообщение в тело компонента. Например, если порт start (пуск) компонента Motor (двигатель) получает стандартное сообщение notify() (оповестить), он передает сообщение start() главному объекту, представляющему двигатель.

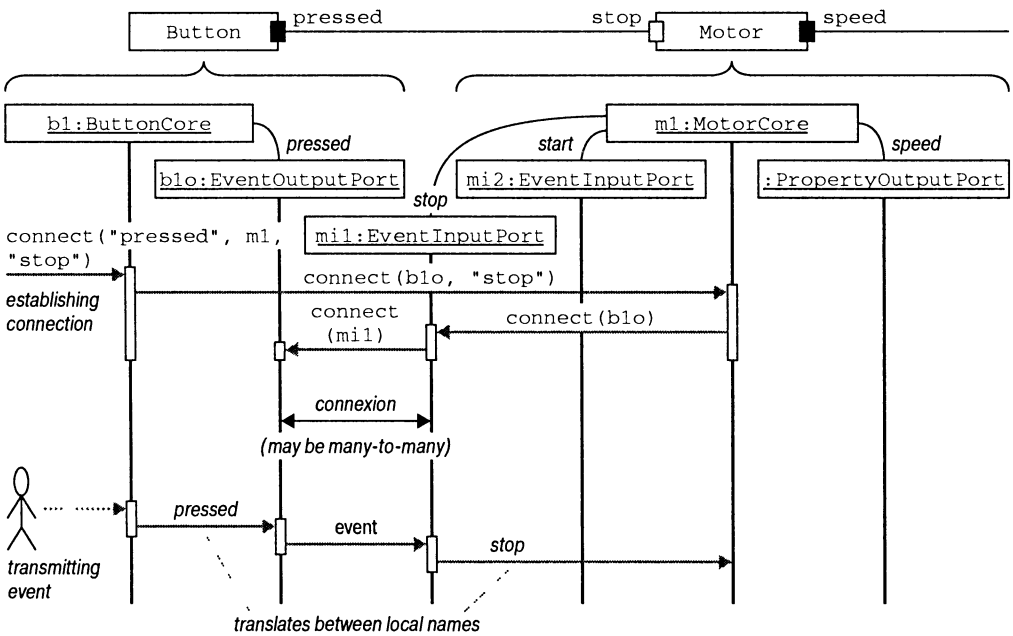


Рис. 7.10. Соединители абстрагируются от протоколов

Выходные порты событий, представленные на этой схеме, передают события при изменении именованных атрибутов объекта-отправителя — иными словами, они реализуют шаблон

Observer. Входные порты свойств обновляют именованный атрибут объекта-приемника. Таким образом, показания измерительного устройства соответствуют скорости работы двигателя (до тех пор, пока они связаны).

Комплект компонентов (component kit) — это набор компонентов со взаимосвязанной архитектурой. Для комплекта определяются соединения — протоколы, которым соответствуют порты. **Архитектура комплекта компонентов** определяет, каким образом работают эти соединения и к какому виду они относятся. Конечно, эти определения должны быть стандартизированы еще до того, как будут написаны непосредственно сами компоненты. Общие типы объектов (`int`, `Plain` и т.д.) должны быть понятны всем членам комплекта. Архитектура комплекта представляет собой базу, на основе которой создаются библиотеки компонентов. Впоследствии приложения могут быть скомпонованы из компонентов, содержащихся в библиотеке, но без архитектуры комплекта вся компонентно-ориентированная разработка (Component-based Development — CBD) рушится. Существует множество архитектур, которые смогут работать с любой схемой соединителей, однако спецификация Java Beans предоставляет немного другой (более надежный) набор протоколов для описанных выше архитектур.

Проектирование архитектуры предполагает принятие проектных решений. Порт должен быть реализован как объект со своими собственными правами. Соединители портов абстрагируются от подробностей протокола порта, но в конечном счете должны быть определены. Например, диаграмма последовательности, представленная на рис. 7.10, показывает только один способ реализации интерфейса "кнопка-двигатель". Аналогичным образом может быть реализован порт соединения свойств, но при условии регулярного обновления значений.

Хотя мы проиллюстрировали принцип соединителей на небольших компонентах, та же идея применима и к большим компонентам, протоколы соединителей которых выполняют сложные транзакции. Текущая работа, связанная с интеграцией приложений для предприятий (Enterprise Application Integration), может рассматриваться как попытка заменить многочисленные протоколы передачи "точка-точка" более немногочисленными универсальными соединителями.

При задании портов (или капсул) определяются типы передаваемых параметров, протокол взаимодействия и язык, при помощи которого может быть выражен протокол. Протокол взаимодействия может быть представлен любым из следующих способов.

- Рандеву языка Ada
- Асинхронный вызов
- Широковещательная рассылка
- Буферизированное сообщение
- Вызов переключения
- Сложная транзакция
- Аналоговый поток данных
- Протокол передачи файлов FTP
- Вызов функции
- Протокол передачи гипертекста HTTP и т.д.

Языком взаимодействия может быть любой из следующих.

- ASCII и др.
- Сообщение или событие CORBA (технологии построения распределенных объектных приложений)
- Вызов динамически подключаемой библиотеки DLL или компонента COM
- Язык HTML
- Протокол Java RMI
- Простой вызов процедуры
- RS232
- Протокол TCP/IP
- Конвейер UNIX
- Язык XML
- Сжатие

7.3. Проектирование компонентов

Один из авторов журнала Bite Джон Уделл (John Udell) выразил свое мнение о том, что "объекты мертвы!" и будут заменены компонентами. Но он написал это в мае 1994 года. В то время объекты действительно не соответствовали основным стандартам информационной технологии, и это продолжалось примерно до 1996/97 года. На самом деле почти все предложения по компонентной разработке, сделанные в 1994 году, не просуществовали слишком долго: я имею в виду OpenDoc, OpenStep, Hyperdesk и другие. Вероятно, единственной значимой технологией была простая технология VBX (управляющие элементы для Visual Basic). В настоящее время существует несколько мнений по поводу того, что означает термин "компонент". Одни специалисты просто применяют этот термин для обозначения любого модуля. Другие подразумевают под ним готовый объект или каркас — единицу развертывания. Третьи считают его двоичным кодом, который может создавать экземпляры (или многочисленные классы). Авторы объектно-ориентированного анализа стремятся отождествить набор интерфейсов с ограничениями, определяющими *предложения* и *требования*. Ограничения, задающие *требования*, часто называют **внешними интерфейсами** (outbound interface). Этот вопрос рассматривается в работе [212], где показано, что подобные аспекты компонента должны быть определены, но они не являются частью соглашения компонента. В [745] компонент определен как "приобретенный и развернутый двоичный элемент независимого производства", а позднее — как "элемент композиции, обладающий только заданными в рамках соглашения (так называемого контракта) интерфейсами и явными контекстными зависимостями". Мы предпочитаем следующее определение: выполняемый элемент развертывания, который принимает участие в композиции.

Во многих отношениях Visual Basic все еще остается парадигмой компонентной разработки. Потребность в компонентах вначале объяснялась тем, что языки объектно-ориентированного программирования были ограничены одним адресным пространством. Объекты, скомпилированные разными компиляторами (даже написанные на одном языке), не могли "общаться" друг с другом, что привело к появлению технологий создания распределенных объектов, например RPC (Remote Procedure Call), DCOM (Distributed Component Object Model), CORBA и т.д. В любом случае определение интерфейсов было отделено от реализации, что делает объектно-ориентированное программирование единственно возможным способом реального кодирования объектов.

В конце 1990-х годов поставщики систем планирования и управления ресурсами предприятий, оказавшиеся в состоянии решить проблему 2000 года и перехода на Евро, сделали хорошее дело, так как их клиенты нуждались в быстрых и исчерпывающих решениях. Когда этот период закончился, их рынок оказался под угрозой из-за возврата к более гибким системам, которые в большей степени были приспособлены к требованиям компаний. Были даже процитированы слова одного поставщика, который говорил, что крупному потребителю следует изменить свои процессы с тем, чтобы они соответствовали пакетам, поскольку метод их работы не соответствует "промышленным стандартам". Подобная самонадеянность уже не могла быть приемлемой после 2000 года, и поставщики, очертя голову, бросились представлять свои предложения в виде компонентов, т.е. все в большей степени приспосабливать их к требованиям заказчика.

Многие разговоры по поводу компонентов, отличавшихся (в лучшую сторону) от объектов, базировались на плохой идее, согласно которой первое место отводилось бизнес-объектам. Многие разработчики полагали, что понятие объекта находится в полном соответствии с семантикой класса или экземпляра языка C++. Мнения других опирались в этом смысле на объектную семантику языка Smalltalk или классов и экземпляров языка Eiffel. Даже те, кто применял классы или экземпляры языка UML, потерпели фиаско, пытаясь предоставить достаточные семантические возможности для этого понятия, как правило, игнорируя наличие правил и инвариантов и не рассматривая пакеты в качестве оболочек (см. главу 6). На самом деле объект будет работать только в том случае, если имеются все необходимые ему службы (серверы). Этой точки зрения такие методы, как SOMA, придерживались уже начиная с 1993 года. Метод SOMA [327] предполагает постоянное наличие пар "сообщение-сервер", являющихся частью описания классов. Они эквивалентны понятию, которое в терминологии компонентной модели объектов COM компании Microsoft получило название внешних интерфейсов.

С одной стороны, внешние интерфейсы нарушают инкапсуляцию, так как компонент зависит от видов сотрудничества, которые могут изменяться и таким образом воздействовать на него. Например, клиент службы управления заказами не должен знать, что этот компонент зависит от компонента управления товарами; в альтернативной реализации все может быть упаковано в единственном объекте. Поэтому в работе [212] утверждается, что виды сотрудничества не входят в соглашение объекта в обычном смысле. Дэниелс делает разграничение между соглашениями использования и реализации. Последние действительно содержат зависимости, поскольку они должны использоваться транслятором приложения. То, что эта область требует дополнительных исследований, было показано на одном из собственных примеров Дэниелса. Рассмотрим финансово-торговую систему, которая взаимодействует с системой предоставления цен. Исходя из приведенных выше рассуждений, пользователь системы не должен беспокоиться о том, осуществляется ли поставка известной и уважаемой фирмой, например компанией Reuters, или фирмой Price, Floggett & Runne Inc. Но, разумеется,

пользователь сильно заботится о надежности информации, и ему следует делать четкое разграничение между этими двумя поставщиками. Этот пример говорит о том, что мы всегда должны учитывать **эффекты** (effect) сотрудничества, рассматривая их как инварианты или наборы правил в соглашениях использования. В этом случае правила касались бы утверждения относительно репутации провайдера информации.

В настоящее время существуют следующие технологии компонентной разработки: CORBA/OMA, Java Beans/EJB, COM/DCOM/ActiveX/COM+, TI Composer и обладающая более академической базой Component Pascal/Oberon. Все эти подходы похожи друг на друга в том, что они концентрируют основное внимание на композиции и перенаправлении данных, составных документах, протоколах передачи данных (например, `.jar`-файлах), соединителях событий (единичных или циркулярных), метаданных и некотором механизме перманентности. К различиям, существующим между этими методологиями, можно отнести их подход к управлению версиями интерфейсов, управлению памятью, следованию стандартам (бинарный, на основе связывания или другой), зависимости от языка и поддержке платформы.

С точки зрения поставщика компоненты обычно крупнее классов и могут быть реализованы при помощи многих языков. Они могут содержать свои собственные метаданные и быть скомпонованы без программирования. Они должны задавать все, что им необходимо для работы. Такие описания могли бы быть едва ли не спецификацией для COM+ и CORBA. Подобные компонентные системы неуязвимы для критики. Размеры компонента зачастую обратно пропорциональны его соответствию любому заданному требованию. Кроме того, компоненты могут тестироваться на поздних этапах разработки, а иногда только после загрузки (хотя аплеты на самом деле не являются истинными компонентами в том смысле, который мы имеем в виду в данном случае). Существует некоторое противоречие между архитектурными стандартами и требованиями, которые могут ограничивать возможности изменения бизнес-процессов. И наконец, существует проблема совместного понимания, затрагивающая интересы разработчиков и пользователей; она будет рассмотрена в главе 8. Зиперски (Szyperski) обстоятельно рассматривает различные критерии решений, применяемые всеми поставщиками инфраструктур и компонентов. Он ничего не говорит о последствиях для пользователей. Можем ли мы сделать вывод, что пользователи не заботятся о том, как разрабатываются компоненты? Конечно, их интересует, как они компонуются и образуют приложения.

Для объектного моделирования не характерно разделение между процессами и данными. Здесь отдается предпочтение инкапсуляции — отделению интерфейсов от реализации. Кроме того, предпочитается полиморфизм — наследование и возможность перекрытия — а также проектирование по соглашению — с использованием ограничений и правил. Принципы объектно-ориентированного подхода могут быть равным образом применены к объектно-ориентированному программированию, объектно-ориентированному анализу, моделированию бизнес-процессов, проектированию распределенных объектов и компонентов. Однако существует различие между концептуальными моделями и моделями программирования. В концептуальном моделировании и компоненты и классы обладают идентичностью. Поэтому компоненты являются объектами. Наследование является таким же фундаментальным принципом проектирования, как и инкапсуляция для разработчика концептуальных моделей. С другой стороны, в моделях программирования компоненты и классы не обладают идентичностью (методы классов обрабатываются экземплярами классов-фабрик). Таким образом, компоненты не являются объектами, и наследование классов (или делегирование) представляет опасность. Поэтому есть ли здесь место для наследования?

Компонентная разработка снижает роль наследования реализаций, но не наследования интерфейсов, однако на концептуальном уровне это разграничение все равно лишено смысла.

Когда приходится связывать модели требований с моделями анализа, можем легко "опуститься" до модели, которая похожа на модель программирования (к чему стремятся почти все методы, основанные на UML), или ввести процесс преобразования моделей (в главе 8, например, предложено преобразование модели SOMA в модель Catalysis). Этот компромисс связан с тем, до какой степени пользователи и разработчики могут обладать общим пониманием.

7.3.1. Компоненты, предназначенные для решения проблем гибкости

Компонентная разработка связана с построением наращиваемых семейств программных продуктов из новых или существующих комплектов компонентов. Диапазон последних очень велик: от отдельных классов до целых (упакованных) существующих систем или коммерческих пакетов. Подобная работа до сих пор считалась труднодостижимой задачей для разработчиков программного обеспечения. Вся тонкость заключается в реализации определений протоколов интерфейсов объектов таким образом, чтобы они могли подключаться различными способами. Количество интерфейсов должно быть небольшим по сравнению с числом компонентов. В целях повышения гибкости эти интерфейсы должны допускать такое согласование, какое характерно для факсимильных аппаратов или интерфейсов в технологии OLE компании Microsoft и т.п.

Для пояснения сути вопроса может быть достаточно одного примера. Рассмотрим систему управления отелями, которая вначале была написана для небольшой цепочки шотландских отелей. Исходная система является очень успешной и быстро окупается. Но цена этого успеха — быстрое расширение, и в настоящее время компания приобретает еще несколько отелей в Англии, Уэльсе, Франции и Германии. В Шотландии комнаты вновь прибывшим всегда выделяются на основании следующих соображений: это должна быть пустая комната, ближе всех расположенная к конторке портье (чтобы предохранить туфли своих посетителей от изнашивания, что неизбежно случается при ходьбе на длинные расстояния). Но расточительный и угрюмый англичанин предпочитает этой экономии тишину и спокойствие. Поэтому комнаты распределяются по-другому: в случае не полностью заполненного отеля пустые комнаты используются для разделения занятых комнат. В Германии вначале заполняются комнаты, имеющие двустворчатые окна до пола. В разных государствах предусмотрены также различные правила выплаты заработной платы. Система постепенно совершенствуется, и вскоре приходится поддерживать несколько версий: одну — с размещением поближе к конторке портье и французскими правилами выплаты заработной платы, другую — с выделением комнат, наименее использовавшихся за последнее время, и английскими законами выплаты зарплаты персоналу, а также специальной подсистемой "заплаты", управляющей рождественскими премиями в Ирландии и Нидерландах, и т.д. В этом и состоит проблема поддержки!

Значительное улучшение в результате некоторой модификации показано на рис. 7.11. Имеется базовая структура, которая охватывает все то общее, что существует между требованиями, предъявляемыми всеми отелями. Каждое изменяющееся требование включено в заменяемый компонент — например, существуют разные распределители комнат, а также разные компоненты выплаты заработной платы сотрудникам. Такая схема намного облегчает

поддержку и управление вариантами системы. Мы разделяем роли разработчика базовой структуры и заменяемых компонентов, которые не могут изменять базовую структуру.

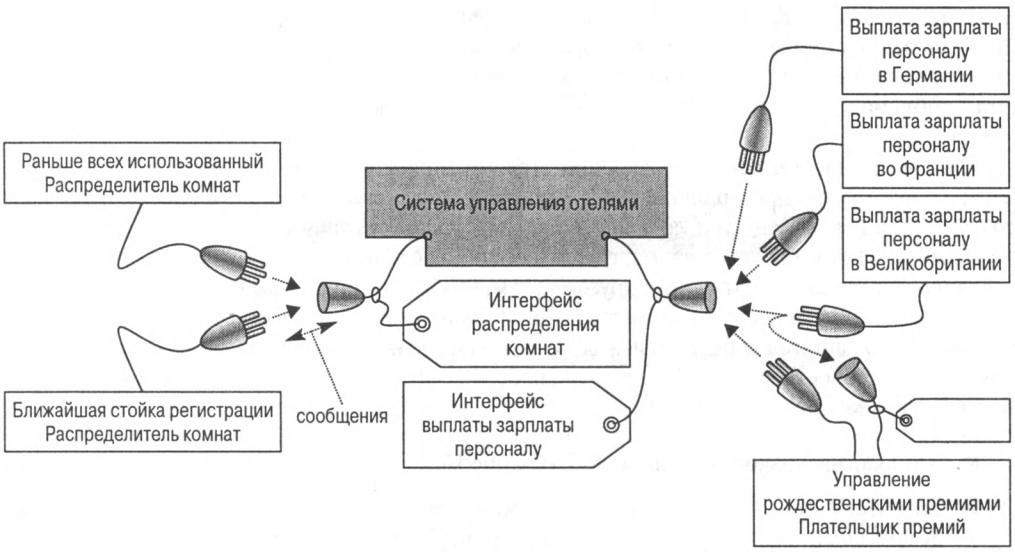


Рис. 7.11. Точки подключения повышают гибкость

При этом становится ясно, что **самый подходящий базис для выбора компонентов состоит в том, что они должны соответствовать изменяющимся требованиям**. Это правило является ключевым, но специалисты иногда его забывают, хотя и утверждают, что занимаются компонентной разработкой.

Одна из проблем состоит в том, что не всегда просто предвидеть, какие требования будут изменяться в будущем. Самый лучший совет, который можно дать, заключается в следующем.

- Придерживайтесь принципа разделения интересов в рамках основной структуры с тем, чтобы можно было достаточно просто заново разложить ее на составляющие.
- Не принимайте во внимание обобщения, в которых вы не уверены (т.е. не уверены, что они вам понадобятся): работа, скорее всего, окажется бесполезной. Следуйте принципу экстремального программирования: "Вам это не понадобится!".
- Если для ввода новых точек подключения понадобится заново разложить базовую структуру на составляющие, делайте за один раз одно изменение и после каждого изменения проводите повторное тестирование.

7.3.2. Крупномасштабные соединители

В предыдущем разделе была рассмотрена идея соединителей, и для иллюстрации этого принципа использовался простой пример: соединители передают простые события и значения-свойства. Но эту же идею можно применить также для случая, когда компоненты представляют собой большие приложения, работающие с собственными базами данных и

взаимодействующие с Internet. Напомним, что огромное преимущество соединителей перед интерфейсами "точка-точка" заключалось в проектировании небольшого числа протоколов, которые являются общими для целого набора компонентов, чтобы они могли быть без труда перегруппированы. В наших небольших примерах это означает, что можно выбирать компоненты из пакета и создавать разнообразные конечные продукты. В больших системах это позволяет легче заново группировать компоненты по мере изменения бизнес-процессов. Эта проблема является общей, и с ней сталкиваются многие крупные и не очень крупные компании.

Например, наша система управления отелями может иметь Web-сервер в Амстердаме, центральную систему резервирования в Эдинбурге, подсистему управления кредитными карточками в Новой Зеландии и локальные системы распределения комнат в каждом отеле по всему миру. И нам бы хотелось определить общий соединитель, общий "язык", на котором все они будут разговаривать друг с другом, с тем, чтобы будущие возможные реконфигурации не предполагали написания множества новых адаптеров. Типичными событиями в наших протоколах соединителей будут приезды и отъезды посетителей, оплачивающих счета, а свойствами — готовность комнат. Архитектура набора компонентов для такой сети должна включать следующее.

- Технологию низкого уровня — COM, CORBA, TCP/IP и т.д.
- Базовую модель бизнес-процессов — объекты каких типов осуществляют связь между компонентами, потребителями, системами резервирования, подготовки счетов и т.д.
- Синтаксис языка, при помощи которого будет описываться модель; часто для этой цели выбирается язык XML.
- Бизнес-транзакции — например, резервирование обеспечивается согласованностью между системой резервирования и местным отелем.
- Бизнес-правила — разрешено ли потребителю иметь комнаты в разных отелях в одно и то же время?

О бизнес-правилах на стадии моделирования иногда забывают. Но важно учитывать следующее: если один компонент системы считает, что клиент может иметь две комнаты, тогда как другой полагает, что каждый клиент должен иметь только одну комнату, то при взаимодействии этих компонентов произойдет путаница. Этот вопрос связан не только со статическими инвариантами: имеет также значение порядок следования операций. Представим, например, компанию, в которую в результате объединения вошли два подразделения, находящиеся в разных государствах. В Великобритании оплата производится до поставки, тогда как в Ирландии — только после поставки. Разные режимы могут быть проиллюстрированы при помощи двух моделей переходов состояний, представленных на рис. 7.12. При взаимодействии этих систем возникнут проблемы, поскольку когда британский клиент заказывает что-то, относящееся к подразделению в Дублине, системы в порядке очередности передадут запрос британской системе. Эта система принимает оплату и поставляет услугу — так что счастливый Джон Бул не будет платить ни пенни. С другой стороны, бедному Пэдди Рили, который делает заказ из лондонского подразделения, предложат заплатить дважды.

Рис. 7.12. Несовместимые бизнес-процессы

7.3.3. Соответствие между бизнес-моделью и реализацией

Метод Catalysis включает специальные приемы проектирования компонентов. Действия уточняются и становятся видами сотрудничества, а виды сотрудничества — портами. Для согласования компонентов с моделью описания применяется метод извлечения (см. главу 6). Рассмотрим ситуацию, показанную на рис. 7.13. Здесь в общих чертах представлены два компонента, предназначенные для ведения учета и диспетчеризации, но они основаны на разных бизнес-моделях. Любая система, содержащая эти компоненты, должна быть основана на типовой модели, которая будет разрешать конфликты имен в базовых компонентах. Например, наша модель должна определять клиента с помощью понятий дополнительного клиента и плательщика. Кроме того, для синхронизации операций необходимо предусмотреть инварианты.

Интерфейсы не могут быть полностью описаны при помощи языков программирования — имеет значение контекст, в котором они используются. Зачастую единственная причина, по которой специалисты принимают списки операций за описания, состоит в том, что их названия уже говорят о предполагаемом поведении. Следует сказать, что компоненты делают то, что они и должны делать, — содержат типовые модели, инварианты, наборы правил и протоколы. Первые три рассматривались в главе 6, а сейчас давайте посмотрим, каким образом можно строго описывать протоколы.

Поскольку соединители должны быть инкапсулированы в объектах, они могут быть специализированы и обобщены. Для задания классов соединителей используются шаблоны или каркасы (см. главу 6). Сначала определяется и задается каждый класс и его интерфейсы. Затем определяются классы соединителей. Далее при помощи диаграмм последовательностей операций и/или состояний можно определить протоколы передачи сообщений для каждого класса соединителя. Необходимо гарантировать, что бизнес-модель, являющаяся общей для всех компонентов, всегда четко представлена в модели типов, обладающей достаточно большими возможностями для определения всех параметров протоколов соединителей. Места потенциальных изменений должны обрабатываться при помощи точек подключения. Бизнес-правила должны быть инкапсулированы в интерфейсах.

Компоненты предназначены для использования в контекстах, неизвестных их проектировщикам, и поэтому должны быть упакованы намного лучше, чем обычные классы в библиотеках. Они должны храниться и поставляться вместе с полной документацией, охватывающей всю спецификацию и интерфейсы, а также порты и их протоколы. Поэтому компоненты

обязательно должны поставляться вместе с соответствующими средствами тестирования. В хороших архитектурах компоненты должны на стадии выполнения отвечать на запросы, касающиеся их соединений. В случае неправильного использования они не должны отказываться функционировать, а скорее должны "выражать недовольство" другому программному обеспечению.

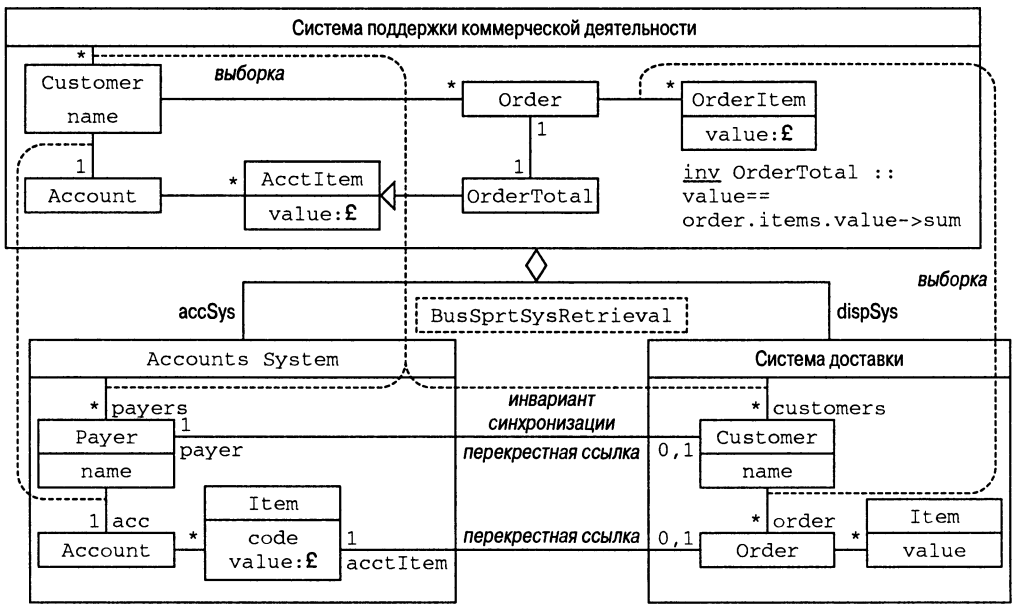


Рис. 7.13. Извлечение модели из компонентов

7.3.4. Бизнес-компоненты и библиотеки

В настоящее время существует три конкурирующие концепции бизнес-объектов. Вероятно, самая первая появилась благодаря Оливеру Симсу [716], который хотел представить сущность при помощи значка, понятного для пользователей. Вторая концепция бизнес-объекта Яна Грэхема (Ian Graham) была независимо разработана примерно в это же время. Он подразумевал под бизнес-объектом любой *объект спецификации*, который упоминается в описании бизнес-процесса и постановке задачи. Третья концепция возникла в рамках дискуссии внутри OMG (рабочей группы по развитию стандартов объектного программирования). Входящая в ее состав группа, занимающаяся вопросами моделирования бизнес-объектов (Business Object Modelling), обнародовала следующую (третью) концепцию бизнес-объекта. Это широко используемый компонент *программного обеспечения*, имеющий отношение к терминологии проблемной области. Это последнее определение приводит к ряду проблем, поскольку оно ограничивает бизнес-объекты только территорией разработчиков коммерческого программного обеспечения и не учитывает стремлений, касающихся возможности моделирования бизнеса, заданий и агентов, а также поддержки многократно используемых объектов независимо от конкретной машины и языка. Концепция бизнес-объекта, предложенная OMG, в ее настоящем виде не позволяет считать бизнес-объектами

"интеллектуальные" агенты, поскольку объекты языка описания интерфейсов IDL не могут с такой же легкостью выражать семантику, как и наборы правил. Однако мы надеемся на будущие изменения в этой области.

Компоненты любого масштаба берут начало от каркасов графических пользовательских интерфейсов, подобных Java AWT, VBX и OCX. Они могут представлять собой фрагменты программы, например Java Beans, и, наконец, целые приложения, преодолевающие языковой барьер и поставляемые в качестве конвейеров UNIX или компонентов COM и OLE (объектно-ориентированной технологии компании Microsoft на основе COM), например Excel. Если мы выйдем за пределы одного компьютера и окажемся в распределенном пространстве, то столкнемся с компонентами, упакованными в соответствии с требованиями технологии CORBA, DCOM и Java RMT. В настоящее время некоторые компании утверждают, что они могут предложить библиотеки бизнес-объектов. Они состоят, главным образом, из библиотек кода. Одна такая библиотека ранее хорошо была известна как модель Infinity и предназначалась для средств анализа финансового рынка. Мы уже упоминали о появившихся во время написания этой книги пакетах планирования и управления ресурсами предприятий ERP (enterprise resource planning), основанных на компонентах. Библиотека San Francisco компании IBM представляла собой одну из первых попыток создания полной библиотеки компонентов и шаблонов проектирования на бизнес-уровне [143, 570]. В Infinity реляционная модель данных, известная как Montage, используется в качестве подструктуры, на базе которой строится объектная модель, реализованная на языке C++. Это приводит к проблемам эффективности, особенно при использовании сложных производных средств, поскольку база данных должна выполнять множество небольших операций объединения для построения сложных составных объектов. Некоторое несовершенство проявляется в том, что в модель не входят инструментальные средства: считается, что пользователи должны сами довести модель до конца. Однако продукт успешно был использован в некоторых банках и вполне подходил для более мелких учреждений такого рода или для тех организаций, которые не работают на рынке крупномасштабных модифицируемых систем. Модель Infinity не поставляет свою абстрактную объектную модель — только предлагает код. Некоторые другие предложения, которые и в самом деле преподносят себя как абстрактные объектные модели финансовых инструментов, обычно (что мы знаем по своему опыту) являются немного бо́льшим, чем просто по-новому обозначенными моделями данных.

Продолжая исследовать сферу финансов как типичную область для моделирования бизнес-объектов, давайте немного отвлечемся и посмотрим, объекты каких типов пригодны (или не пригодны) для моделирования их в виде бизнес-объектов. Класс финансовых средств, как оказалось, является вполне подходящим для этой цели, поскольку каждый такой инструмент имеет ясное, постоянное и однозначное определение. Однако на самом деле существует свыше 300 классов даже в самой грубой модели финансовых средств. Это означает, что построить стабильную структуру классификации чрезвычайно сложно, даже допуская использование разделения, выполняемого при помощи интерфейсов. Более того, многие финансовые продукты являются составными. Все это приводит к тому, что инструменты могут моделироваться двумя совершенно различными способами. Упрощенный способ состоит в том, чтобы создать глубокую структуру классификации, отмечая структуры композиции каждого инструмента по мере их появления. Мы построили такие модели и считаем их полезными, если они не очень велики. Этот подход применяется в модели Infinity. Вероятно, более элегантный подход основан на теореме финансового проектирования, которая гласит, что каждый инструмент состоит из фундаментальных элементов и их возможных вариантов. Это означает, что, не полагаясь исключительно на структуру классификации, можно также использовать композицию.

Этот подход применяется, по крайней мере, в одной важной и успешно работающей системе управления рисками в банке Chase Manhattan Bank. Однако эти общие проблемы классификации все еще остаются очень сложными и поэтому не должны решаться неопытными специалистами по моделированию. Тем не менее небольшие структуры классификации вполне пригодны для включения в вашу первую библиотеку, содержащую многократно используемые компоненты.

Корпоративные действия представляют собой очень хороший пример и идеально подходят для рассмотрения бизнес-объектов. Существует ровно 42[8] таких действия, и они могут быть легко классифицированы. На рис. 7.14 представлен фрагмент такой структуры классификации. Нетрудно заметить, что даже эта структура является достаточно сложной. Представьте себе, как должна выглядеть структура, включающая 350 классов!

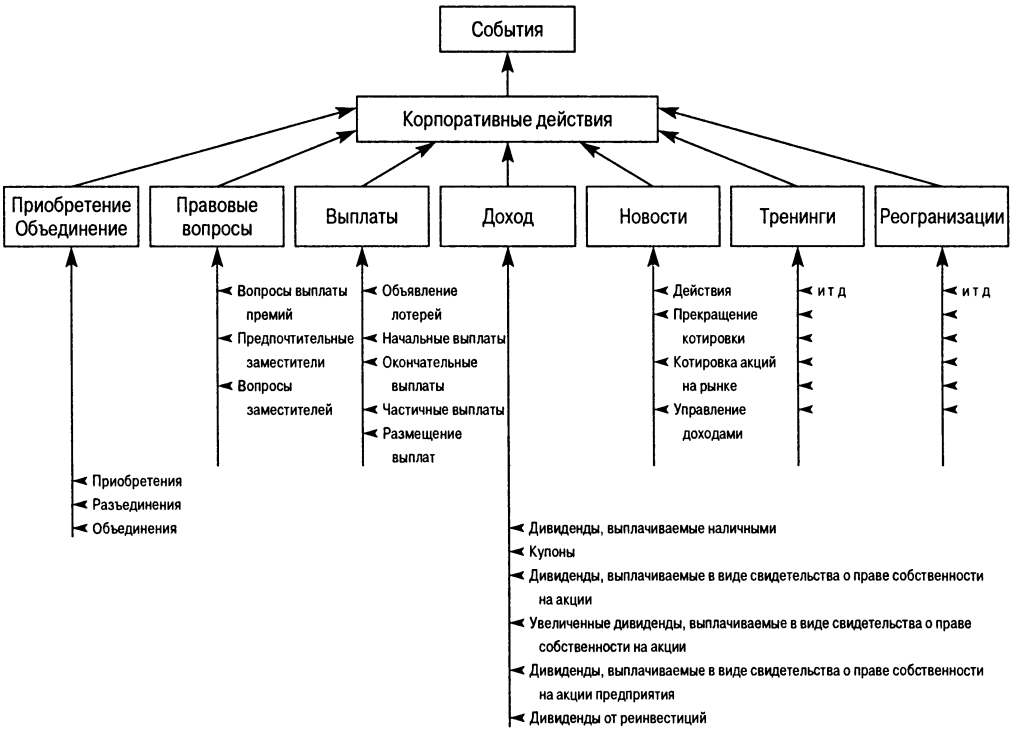


Рис. 7.14. Классификация корпоративных действий

На первый взгляд, алгоритмы ценообразования не кажутся вполне подходящими для реализации в виде бизнес-объектов, поскольку они представляют собой чистые функции, тогда как настоящие объекты, конечно, должны иметь также и данные. Рассуждая таким образом, можно прийти к выводу, что товар должен знать, как оценивать самого себя. Однако в области финансов существует огромное множество способов оценки одного и того же товара. Все эти алгоритмы развиваются по мере того, как в финансовой области появляются новые

[8] Возможно, отвечая на вопрос, поставленный Дугласом Адамсом (Douglas Adams).

открытия. Поэтому не стоит упаковывать все эти алгоритмы в класс `Products` (товары). В определенном случае алгоритмы должны быть реализованы в виде объектов. Они, конечно, представляют собой не более (но и не менее) чем библиотеки функций, представленные в объектно-ориентированной форме. Но почему бы и нет?

Такие события, как продажи, оплата, движение наличности и т.д., связаны с еще большими проблемами. Для продаж характерны сложные жизненные циклы, которые, возможно, могут быть представлены при помощи конечных автоматов. Можно выделить следующие состояния: оценки, согласования, утверждения, подтверждения, приведения в соответствие и т.д. Сложность заключается в том, что разные средства имеют различные жизненные циклы: установка курса иностранной валюты в достаточной степени отличается от установки курса ценных бумаг. Значит, нужно либо попытаться спроектировать бизнес-объект `Trades` (торговля), охватывающий все возможные случаи, либо создать разные объекты, предназначенные для различных операций. Наша интуиция проектировщиков объектов подсказывает, что единственный класс `Trades` должен ссылаться на сложную иерархию средств, где каждый инструмент отвечает за информацию о собственном цикле жизни. Однако мы уже говорили, что иерархия инструментов является чрезвычайно сложной, так что подобное ее построение является непривлекательным. На практике такое решение представляется весьма трудным и должно приниматься в соответствии с обстоятельствами и по усмотрению заинтересованных лиц.

Что можно сказать о клиентах, специалистах по управлению и т.д.? Они являются наихудшими кандидатами на бизнес-объекты, которые только можно себе представить, поскольку не существует такой сущности, как клиент. Взгляните на компьютерные системы вашей компании. Скорее всего, каждая система имеет свое определение клиента. И это не потому, что все прошлые поколения проектировщиков были глупцами. Это объясняется тем, что каждое подразделение имеет **разные** отношения со своими клиентами. Например, в одном случае клиенты — это *роль*, признаваемая юридическим лицом. Точно так же можно сказать, что клиент — это вид *отношения* между вашей организацией и другой фирмой. Таким образом, ваша библиотека бизнес-объектов должна иметь то, что Мартин Фовлер назвал бы классом `Parties` (стороны), а также группы отношений, представляющих понятия типа "клиенты по расчетам", "заграничные клиенты" и т.д.

Бизнес-процессы часто представляют в виде бизнес-объектов. Мы полагаем, что это и правильно, и неправильно. Если реальные программные объекты представляют процесс, они годятся для отображения только самых простых процессов, и мы по своему опыту знаем, что в этом случае могут получиться чрезвычайно жесткие последовательности выполнения действий. Тем не менее подход, описанный в следующей главе, действительно использует объекты (агенты, диалоги и действия) для моделирования бизнес-процессов, хотя и косвенным образом, что позволяет увеличить гибкость и выразительность. Поэтому следует опасаться людей, продающих "объекты бизнес-процессов".

В настоящее время существует несколько концепций модели бизнес-объектов. Мы рассмотрим одну из них как модель понятий предметной области, выраженную в терминах объектов. Конечно, в данном представлении это не модель коммерческой деятельности и ее процессов. Это также и не полная модель системы (до тех пор, пока она не уточнена благодаря проектированию логики). Некоторые коммерческие предложения, названные "вертикальной моделью бизнес-объектов", представляют собой просто немного усовершенствованные и поновому обозначенные модели данных и процессов, использующие множество ложных предположений о бизнес-процессах, которые были получены от немногочисленных сторонних организаций. Таким образом, лучшим решением может быть создание собственной модели

бизнес-объектов, в результате чего вы получите разработанный по индивидуальному заказу ресурс и усилите свой контроль над ситуацией. Такая стратегия позволяет в случае необходимости оптимизировать эффективность, например, посредством выбора нужной технологии для баз данных. Вы сможете воспользоваться преимуществами новых технологий по мере появления таких новшеств. Однако собственная технология может стоить очень дорого.

Бизнес-объекты были громко разрекламированы. При рассмотрении вопроса о покупке коммерческой модели бизнес-объектов нужно выяснить, подходит ли эта модель для вашего бизнеса или вы должны будете приспосабливаться к ее недостаткам. Особенно остерегайтесь моделей данных, которые маскируются под объектные модели. Их можно встретить повсюду. Остерегайтесь также моделей, полученных в виде готового кода, особенно в том случае, если реализация выполнена на некотором языке, не являющемся объектно-ориентированным, например на PowerBuilder или Visual Basic. Попросите поставщика предоставить вам историю продукта. Если модель реализована, спросите также, как реализована и спроектирована база данных: соответствует ли ее эффективность типу и масштабу вашего приложения. Используйте двоичный код только тогда, когда вы ему доверяете. Приобретите или создайте подходящее промежуточное программное обеспечение; будьте готовы заплатить за него. Архитектура, основанная на правильных решениях, принятых в отношении промежуточного программного обеспечения, является базисом для преимуществ многократного использования.

Компонентная разработка базируется на архитектуре и оболочках. Об этом со всей очевидностью свидетельствуют новые методологии проектирования, например метод Catalysis. Однако компонентный анализ иногда выглядит как оксюморон (сочетание противоположных по значению слов — *примеч. пер.*). Необходимо осуществить соответствующий переход от надежных технологий объектно-ориентированного анализа к компонентному проектированию, в ходе которого из моделей процессов могут быть получены прецеденты и действия. Согласованность модели и реализации обеспечивают методики "уточнения", используемые в рамках Catalysis.

Пока остается неясным, имеется ли на самом деле развитый рынок компонентов. Существуют специализированные элементы управления OCX, Java Beans и даже San Francisco. Но, вне всяких сомнений, диапазон и количество типов продаваемых компонентов будут увеличиваться. По мере того как это будет происходить, мы все больше и больше будем отделять создание компонентов от компоновки приложения. Оба эти вида деятельности нуждаются в новых моделях процесса разработки. Кроме того, появится необходимость в методологических изменениях.

В новых методах наследование должно по-разному рассматриваться на разных уровнях: концептуальном и уровне компонентов. Остается фундаментальное понятие, предназначенное для концептуального моделирования и представления знаний; его без труда могут понять пользователи. Однако наследование, как мы уже видели, — это не идеальный способ придания гибкости компонентам. Представленные выше точки подключения на основе шаблонов обеспечивают намного большую гибкость. Конечно, мы немного подвергаем опасности незаметность для пользователя, но "овчинка стоит выделки". Аналогичным образом при проектировании гибких компонентов ключевым является принцип заменяемости, но во время концептуального моделирования он зачастую противоречит интуиции.

Компонентную разработку часто противопоставляют объектно-ориентированной, как будто они совершенно различны. Одна из причин, объясняющих такое положение дел, состоит в том, что в некоторой (плохой) практике при объектно-ориентированном программировании акцентируется внимание на очень небольших объектах в ущерб объектам, представляющим

уровень предприятия, и полностью игнорируются виды сотрудничества (внешние интерфейсы) и семантика интерфейсов (правила и т.д.). Главное внимание уделялось слабым объектным моделям, основанным на языках, а не концептуальным моделям, обладающим большими семантическими возможностями, сторонниками которых всегда были такие методы объектно-ориентированного анализа, как SOMA. В этом смысле компонентная разработка просто свидетельствует о том, что некоторые из нас неправильно понимают объектно-ориентированный подход.

Кроме того, можно заключить, что успешная компонентная разработка потребует акцентирования внимания на архитектуре и шаблонах, а также способах моделирования объектов, обладающих широкими семантическими возможностями, которые предоставляются такими методами, как Catalysis и SOMA. Организация, переходящая на компонентную разработку, всегда должна заранее рассматривать архитектуру и вопросы многократного использования. Кроме того, она должна управлять процессом многократного использования. Этот вопрос будет рассмотрен в главе 9. Основные методологические изменения вносятся на уровне проектирования, архитектуры и реализации.

Не забывайте, что модели являются мощным представлением знаний, а не только продуктом компьютера! Объектные модели могут использоваться для представления многих сущностей, т.е. любого вида знаний, касающихся объектов. Поэтому семантика используемого языка моделирования должна быть достаточно мощной, чтобы выражать это знание. Это означает, что семантика должна включать инварианты классов (в идеале — наборы правил), взаимосвязи использования и другие характеристики, необходимые для моделирования бизнесс-процессов. Это объясняется тем, что простого компьютерного моделирования никогда не бывает достаточно. Хорошая модель должна предусматривать диалоги с пользователем, соглашения, цели и задания. Целиком и полностью сами осуществляйте контроль над спецификацией, а не полагайтесь во всем на внешних поставщиков.

Помните также, что совершенно новые системы — это исключения; большинство компонентов будет использоваться в контексте уже существующих систем, и они должны быть спроектированы как компоненты, относящиеся к уровню проектирования. Во многих случаях появится огромная потребность в репозитарии, содержащем существующие коммерческие компоненты, а также предшествующие спецификации (процессы, объекты и т.д.). Должен существовать относительно постоянный архитектурный каркас.

7.4. Резюме

В этой главе приводится анализ концепций архитектуры программного обеспечения, шаблонов и проектирования компонентов. Мы показали, что архитектура является не просто структурой — она должна включать представление и логическое обоснование. Кроме того, мы показали ее значимость.

Мы познакомили вас с некоторыми шаблонами проектирования и анализа, а также архитектурными и организационными шаблонами и затронули идею языков шаблонов и связанных с ними понятий, например, рамок проблемы.

Создавая модели на такой основе, мы широко использовали шаблоны и понятия, характерные для метода Catalysis, например метод извлечения модели, чтобы показать, как нужно проектировать надежные и гибкие компонентные системы. Основное внимание было уделено способам разделения компонентов. И наконец, были рассмотрены некоторые вопросы, связанные с появлением рынка компонентов.

7.5. Дополнительная литература

В работах [59, 108, 704] архитектура рассматривается как структура. Серия статей Алана О'Каллагана (Alan O'Callaghan) в журнале *Application Development Advisor* представляет иную точку зрения, более близкую к той, которой придерживаются авторы этой главы.

Шаблоны проектирования впервые были описаны в [291]. В [133] эти шаблоны приводятся вместе с архитектурными шаблонами. Работа [641] является одной из первых в этой области и знакомит читателя с идеей меташаблонов. В [281] рассматриваются шаблоны анализа. Эта идея еще раньше была предложена Коадом [167]. Ежегодные труды конференции PLoP [194, 771] содержат множество интересных толкований и новых шаблонов. Организационные шаблоны впервые описаны в [192]. Работа [62] представляет собой превосходное описание того, как использовать шаблоны в контексте языка Smalltalk. Она заслуживает прочтения даже теми, кто не интересуется этим языком. В [192] описаны идиомы языка C++. В работе [573] исследуются шаблоны, предназначенные для реализации технологии CORBA. Многие современные работы, посвященные шаблонам, появились благодаря книгам Кристофера Александера и его коллег [21–23], относящимся к области строительства. В [474] дается краткое изложение идей Александера. Книга Ричарда Габриэля [287] также демонстрирует глубокое понимание значимости идей Александера.

В [507] рассматриваются различные концепции шаблонов в контексте "ориентированного на шаблоны" языка BETA, что делает шаблоны обобщениями классов. Но все еще неясно, какое отношение такая концепция имеет к шаблонам, представленным в этой главе.

В [413] описываются шаблоны представления требований, т.е. чтобы быть более точными, анализируются проблемы разработки программного обеспечения, а не решения. Автор расширяет границы своей же трактовки, изложенной в более ранней своей работе [411].

В [293] обсуждается идея когнитивных шаблонов, основанных на шаблонах заданий, характерных для методологии представления знаний Common KADS . Идея заключается в том, что стратегии решения общих проблем, например, диагностики, планирования и выбора товара могут классифицироваться как шаблоны логического вывода.

В [123] рассматриваются **антишаблоны** (anti-pattern) — общие решения часто возникающих проблем, которые *не* работают — и предлагаются свои решения. Эта идея во многом схожа с "заболеваниями программного обеспечения", описанными в [425].

Метод OORAM [654] представлял собой известную технологию, инструментальное средство и язык, в котором виды сотрудничества и их композиция являются центральными механизмами проектирования. Этот метод кратко рассматривается в приложении Б.

В [204] описан метод Catalysis, предназначенный для компонентной разработки. Работа [792] представляет собой (мы надеемся) хорошо написанное введение в этот метод. В книге [158] приводится описание простых процессов определения компонентов программного обеспечения и систем, основанных на компонентах. В этих процессах используется система обозначений UML и идеи, заимствованные из метода Catalysis. Значительная часть разделов 7.2 и 7.3 основана на техническом описании TriReme Алана Уиллса (Alan Wills).

7.6. Упражнения

1. Выберите из приведенного ниже списка два архитектурных стиля и сравните их: многоуровневая структура, методология "классной доски", конвейеры и фильтры, CORBA, соединение равноправных узлов, клиент/сервер.
2. Дайте определение архитектуры программного обеспечения. Почему нельзя просто считать архитектуру структурой высокого уровня?
3. Дайте определение терминов: шаблон, шаблон проектирования, шаблон анализа, организационный шаблон, архитектурный шаблон. Приведите примеры каждого вида шаблонов.
4. Чем отличается каталог шаблонов от языка шаблонов? Свой ответ проиллюстрируйте примерами.
5. Напишите шаблон в стиле Александера, показывая при этом, каким образом отделить фигуры, нарисованные в графическом и стандартном редакторе. Выполняя задание, рассмотрите "сглаженные" и "несглаженные" многоугольники, а также проблемы увеличения или уменьшения числа их вершин.
6. Напишите шаблоны для следующих ситуаций:

   а) нужно проинформировать всех пользователей электронной почты об изменении их адресов, даже если некоторые из них находятся в отпуске;

   б) пользователи, принимающие участие в семинарах, постоянно не согласны друг с другом;

   в) управление противодействием объектной технологии, так как пользователи считают ее слишком опасной.

7. Рассмотрите все преимущества и недостатки наследования по сравнению с делегированием.
8. Рассмотрите различия между методом Catalysis или другой известной объектно-ориентированной методологией и компонентной разработкой.
9. Реализуйте следующий мини-проект.
10. Создайте шаблон каркаса для процесса составления школьного расписания: выделение подходящих классных комнат и компетентных учителей для десяти предметов и пяти уроков продолжительностью в один час по будним дням. Включите в проектное решение все инварианты. Например, предмет химии должен проводиться в комнате, имеющей раковину; в неделю должно быть не менее пяти часов математики и английского языка. Существуют ли здесь какие-либо правила, которые сложно выразить на языке объектных ограничений OCL? Почему это так? Примените этот же каркас для планирования производства на предприятии.

8

Инженерия требований

Существует много причин, побуждающих романистов взяться за перо, но у них всех есть общее — потребность создания альтернативного мира.

Джон Фоул (газетная статья)

Большинство подходов к объектно-ориентированному анализу и проектированию не учитывают вопросы инженерии требований и не рассматривают проблему в целом. Предполагается, что для этого достаточно языка UML, который мы исследовали в главах 6 и 7, и средств трассировки, основанных на прецедентах и текстовом процессоре. В этой главе мы увидим глубокие теоретические и практические причины, по которым такой подход не может быть одобрен. Мы также узнаем, что эти вопросы связаны с процессом обратного проектирования и различными вариантами развития организации. Затем будет описан систематизированный подход к объектно-ориентированному моделированию бизнес-процессов и разработке требований SOMA, который полностью соответствует современным методам анализа типа Catalysis и может использоваться со стандартными методами и инструментальными средствами UML. По ходу мы сможем рассмотреть некоторые проблемы в изучении этой быстро развивающейся области.

8.1. Подходы к инженерии требований

В [633] инженерия требований определяется как "систематический процесс разработки требований с использованием итерационного кооперативного процесса анализа проблемы, документирования результатов наблюдений в различных форматах представления и проверки точности полученного понимания". Такое определение, хотя оно и оставляет некоторые вопросы без ответа, является хорошей отправной точкой, так как предполагает, что существует гораздо больше требований к разработке, чем записано в функциональной спецификации. В отличие от приведенного определения, в работе [233] требования описываются следующими способами.

1. Условия или мощности, необходимые пользователю для решения проблемы или достижения цели.
2. Условия или возможности, которые должны быть выполнены или которыми должна обладать система или компоненты системы, чтобы выполнить требования соглашения, стандарта, спецификации или других формально прилагаемых документов.
3. Документальное представление условий или возможностей, указанных в п. 1 или 2.

Из контекста вытекает необходимость разработки документа технических требований и включения разработки технических требований в контракт.

В [499] указано, что определение Пола (Pohl) поднимает ряд важных вопросов, а именно: можно ли систематизировать нечетко сформулированные требования, можно ли определить полноту требований в контексте итерации, как описать сотрудничество агентов, какое формальное представление можно использовать и, наконец, как можно достичь подлинно общего понимания. Метод, рассматриваемый в этой книге, дает ответы на все эти вопросы в контексте объектно-ориентированной разработки.

Формально или неформально

Различается два аспекта инженерии требований: выявление (формулировка) необходимых требований и их анализ. В процессе выявления необходимых требований компания-разработчик выясняет, что необходимо и почему. Этот процесс относится к дисциплине извлечения необходимых знаний и требует применения многих методов, относящихся к этой дисциплине. Анализ требований, с другой стороны, представляет собой процесс понимания существующих или выявленных требований. Здесь разработчик требований задается вопросами о полноте и структуре полученной информации. Это различие позволяет разделить огромный труд по разработке технических требований на две довольно различные части. Одна часть концентрирует внимание на методах получения знаний и основывается на теории влияния человеческого фактора, гуманитарных системных методах и эргономике. Вторая часть акцентирует внимание на формальных методах системного анализа. Примеры таких методов варьируются от традиционных подходов к системному анализу, например метод JSD [410], до формальных математических методов типа VDM и Z. В контексте объектно-ориентированных разработок существовало несколько попыток распространить формальные теории на объектно-ориентированный анализ.

Одна из проблем формальной спецификации заключается в том, что настоящее требование может быть легко проигнорировано как неформальное и ориентированное только на документ. Вот характерный пример. Печально известный случай произошел при проектировании

самолета. Требованиями устанавливалось, что двигатель малой тяги не должен включаться, пока самолет не коснется взлетно-посадочной полосы. По мнению проектировщиков, это должно происходить при посадке самолета, когда его колеса начинают вращаться вперед, что бывает тогда, когда происходит контакт со взлетно-посадочной полосой или ей подобной. Поэтому они построили систему таким образом, что двигатели малой тяги включались автоматически, когда между колесами и полосой не было зазора. Это решение работало очень хорошо, пока однажды самолет не совершил посадку на скользкую полосу, покрытую водой! Самолет перелетел взлетную полосу. Поэтому, даже если система может быть подвергнута проверке на соответствие спецификации (почему бы это не сделать в нашем примере с самолетом?), все же нет никакой гарантии, что спецификация соответствует истинным требованиям. Это случается главным образом тогда, когда существуют противоречия в требованиях. В только что приведенном примере противоречие заключалось между потребностью устранения ошибки человека и необходимостью безопасной посадки самолета в любых условиях.

Формальные методы позволяют проверить соответствие кода спецификации системы. Это достигается путем написания спецификации на языке, основанном на некотором варианте формальной логики. Затем может быть построено математическое доказательство корректности. В связи с этим возникает две основные проблемы. Во-первых, доказательство требует высокой математической квалификации и для больших систем может быть довольно затруднительным. С другой стороны, для систем безопасности и ключевых подсистем, от которых зависят другие жизненно важные системы, усилия могут быть оправданы. Один из самых больших проектов такого типа относится к спецификации монитора транзакций CICS. Во-вторых, что, по моему мнению, более губительно, использование логики в качестве языка спецификации равносильно программированию на логическом языке. Формальными методами можно доказать эквивалентность двух программ; однако эти методы ничего не говорят о выполнении требований пользователей. Как мы увидим, метод, представленный в этой главе, направлен на решение этой проблемы.

Примеры объектно-ориентированных подходов к формальной спецификации включают технологию Object Z [145, 240], основанную на нем работу Поля Светмана (Paul Swatman) над языком моделирования FOOM [742]. В этой работе формальный язык используется для выражения соглашений в форме логических утверждений. Catalysis [204] — еще один метод, который подчеркивает формальность, но без обширного использования математических обозначений. В методе Syntropy [188] для достижения подобных результатов применяются диаграммы состояний. Все эти подходы, в основном, относятся к анализу требований, которые уже известны и поняты, а также к точно определенным системам. Эти методы, по своей сути, не подразумевают процесс получения знаний.

Подобный комментарий относится и к подходу, описанному в [788]. Он подразумевает анализ требований с использованием функциональной декомпозиции, моделей сущность-связь и метода JSD и направлен на написание спецификации требований. Этот подход по своей сути является расширением метода инженерии информации (или SSADM) и, как таковой, не подходит ни для объектно-ориентированного, ни для эволюционного методов разработки.

В [411] подчеркивается использование проблемных фреймов для понимания и анализа требований. Этот метод является фундаментальным, но в нем человеческому фактору уделяется гораздо больше внимания, чем при традиционном анализе требований. Он делает ударение на формальной стороне, но без использования чуждой символики, заложенной в традиционных формальных методах. Самое важное, на что указывает Джексон (Jackson), состоит в том, что универсальность метода обратно пропорциональна его применимости. Поэтому он предлагает построить библиотеку проблемных фреймов, охватывающую часто встречающиеся

типы проблем, и разработать соответствующие методы работы с ними. Проблемные фреймы уже обсуждались в главе 7 в контексте шаблонов.

ETHICS

В противоположность формальным и полуформальным методам, в таких подходах, как ETHICS [578], особое значение придается социальным и техническим аспектам системы. Метод ETHICS (Effective Technical and Human Implementation of Computer-based Systems — эффективная техническая и человеческая реализация компьютерных систем) поддерживает двенадцать основных шагов.

1. Определение цели работы (миссии).
2. Описание текущих видов деятельности и потребностей.
3. Определение шкалы измерения оплаты труда.
4. Решение о необходимых изменениях.
5. Определение задач, необходимых для повышения эффективности и производительности труда.
6. Рассмотрение возможных организационных вариантов.
7. Реорганизация.
8. Выбор компьютерных решений.
9. Обучение штата.
10. Работы по перепроектированию.
11. Реализация.
12. Оценка.

Таким образом, метод ETHICS больше напоминает современные средства перепроектирования бизнес-процессов и анализа последовательности выполняемых работ и меньше связан со строгими бюрократическими системами, подобными реализованной в Lotus Notes и т.п. Мне кажется, что при разработке требований и в процессе системного анализа должны быть выполнены все двенадцать шагов, но не обязательно в указанном порядке. Неявно ETHICS подразумевает командную работу и использование формального инспектирования для содействия самоисправлению дефектов. Главная проблема метода ETHICS состоит в том, что он не дает никаких рекомендаций по моделированию, что является критическим фактором успеха шага 8. При этом он не интегрирует идеи перестройки бизнес-процессов, быстрой разработки или объектной технологии. Таким образом, не игнорируя преимуществ ETHICS, двинемся дальше.

Социально ориентированные методы

Методы этнографии и этнологии, вытекающие из антропологии, применяются в инженерии требований для анализа задач с учетом социального контекста. В течение некоторого времени изучается поведение групп и делается заключение относительно требований. В [738] приводится прекрасный пример: применение метода к проектированию фотокопировального аппарата. В данной главе мы адаптировали этот подход, уделив основное внимание сетевым и общественным взаимосвязям, не требующим больших наблюдений.

Главной особенностью рассматриваемого подхода является привлечение пользователей к определению технических требований и проектированию. Основные рекомендации даны в [250, 252]. Здесь требования не фиксируются в некоторых произвольных точках, как это делается в обычных структурных подходах. Этот взгляд согласуется с эволюционным моделированием процесса, описанным в главе 9. Важным преимуществом этого метода проектирования является то, что создание системы не приведет к деградации или снижению профессионализма пользователей. В [777] представлено несколько довольно шокирующих контрпримеров в контексте ухудшения работы с феминистической точки зрения, которые привели, как следствие, к ухудшению или невозможности совместной работы мужчин и женщин. Мне кажется, что при проектировании систем должны быть приняты во внимание все важные социальные факторы (включая, возможно, возраст, культуру, пол и физические способности) и что проектировщики социально ответственны, по крайней мере, за прогнозирование результатов их технологии.

Исследование взаимодействия человека с компьютером (HCI — Human-Computer Interaction) привело к методам проектирования, направленным на пользователя и задачу. Они могут предполагать прямое участие пользователя, анализ наблюдений или даже анкетные опросы и отчеты. В [244] представлен подход к проектированию с ориентацией на пользователя, который имеет много общего с ETHICS, хотя и делает большее ударение на общественно-технические решения и анализ затрат. Подход с ориентацией на задачу присущ технологиям 1950-х и 1960-х годов. Анализ задач пользователя является основой подхода, представленного в этой главе, но мы объединяем эти идеи с применением прецедентов, семиотики (часть этнометодологии) и теории сценариев из искусственного интеллекта. Метод SOMA предусматривает использование ряда других приемов, таких как решетки Келли (Kelly), связанные с инженерией знаний. В главе 6 было уже представлено несколько таких методов.

В [73] описан метод контекстных запросов, основной принцип которого заключается в том, что бизнес-процессы лучше всего постигаются в контексте рабочего места. Он уходит корнями в работу, выполненную компанией DEC в начале 1990-х годов. Данный метод предполагает создание моделей потоков, сценариев задачи, используемых артефактов с учетом культуры и физической среды. Он не ориентирован на проектирование интерфейса пользователя и является более подходящим для автоматизации существующих ручных процессов, чем для перепроектирования.

Проектирование, ориентированное на использование [184], строится на многих упомянутых выше идеях и делает ударение на разработке интерфейса пользователя, анализе задачи, ролях пользователя и основных прецедентах (см. раздел 8.6). Он имеет много общего с представленным здесь подходом SOMA.

Метод функции качества (или так называемый дом качества) обеспечивает способ выявления требований пользователей с учетом свойств разрабатываемого продукта. Мне кажется, что методы, описанные в этой книге, делают этот подход излишним.

Метод CREWS инженерии требований [512] основывается на сценариях проверки согласованности и, что более важно, полноты спецификации. Эта важная работа обсуждается дальше в разделе 8.10.

Метод анализа последовательности выполняемых действий, разработанный в [800], содержит много свойств, которые можно рекомендовать к использованию и интегрировать в наш подход. Он ориентирован на сетевые взаимосвязи между агентами производственной деятельности и их последовательное общение. Однако этот метод критиковался из-за того, что в его контексте выдвигались слишком жесткие технические правила. Мне кажется, что

это можно преодолеть за счет более глубокого изучения бизнес-процессов и учета целей заинтересованных лиц. Более подробно этот вопрос будет рассмотрен в разделе 8.5.

ORCA

Метод ORCA (Object-Oriented Requirements Capture and Analysis — объектно-ориентированный сбор требований и анализ) [503] представляет одну из немногих попыток сделать инженерию требований для объектно-ориентированных разработок более строгой. Этот метод предусматривает использование в качестве отправной точки "детальной картины" программных систем, а затем, подобно методам Catalysis, SOMA и Syntropy, делает четкое различие между моделями мира и моделями проектируемых систем. Подход ORCA основан на разграничении *целевых* и *поведенческих* сущностей и обеспечивает две отдельные системы обозначений для представления каждого аспекта модели. Целевые сущности представляют собой не прямо наблюдаемые объекты, такие как страны, а соответствуют нашим агентам, которые будут обсуждаться в разделе 8.5. Поведенческие сущности соответствуют классам UML. В [503] подчеркивается, что эти два типа сущностей могут совпадать, как в случае организационных структур. Здесь имеется соответствие с классами агентов метода SOMA, которые описывают (или "отражают") реальные агенты в компьютерной системе. Ясно, что метод ORCA имеет некоторые общие точки с SOMA, в том числе в решении комплексной проблемы инженерии требований, в отличие от большинства других объектно-ориентированных методов. Однако существуют различия, среди которых неспособность ORCA обеспечить формальную связь между его двумя языками моделирования и его подход к моделированию на основе диаграмм "сущность-связь". Например, в ORCA допускаются двунаправленные связи и даже ребра исключений в стиле SSADM.

Метод ORCA ориентирован на перепроектирование бизнес-процессов и среду внедрения системы. В этом смысле это ново и мощно. Однако его использование в методах объектного моделирования до некоторой степени прозаично. Семинары и быстрая разработка не предполагаются, но цели бизнес-процесса обсуждаются как часть процесса извлечения знаний.

Модель мира ORCA состоит из объектов, которые представляют целевые сущности и существующие между ними соглашения, выраженные в терминах услуг с пред- и постусловиями и ограничениями (инварианты классов). Эти классы могут также включать нефункциональные требования. Из этой модели интуитивно получается поведенческая (традиционная объектная) модель, которая затем описывается с использованием расширенных диаграмм "сущность-связь" и диаграмм, которые, по существу, являются расширением диаграмм последовательностей UML, хотя и с более ясной семантикой и меньшей ориентацией на программный код. Основной вклад ORCA — обеспечение формального синтаксиса для обоих языков моделирования. Это дает возможность легко проверять соответствие моделей и выявлять ошибки, откуда следует строгость метода ORCA. Однако ORCA не позволяет, несмотря на строгость, доказать, что его модель системы отвечает требованиям, установленным в модели мира. Позже из этой главы будет видно, что возможность делать это точно лежит в основе строгости SOMA, хотя SOMA в настоящее время не имеет формального синтаксиса.

Метод SSM

Исследование программных систем [156, 157] подразумевает целостное изучение поставленной проблемы в контексте организации и решения задачи. Основная задача сводится к выделению поведения и динамики из множества возможных вариантов. В основу работы

программных систем положена общая теория систем и кибернетика. Эти же методы лежат в основе перепроектирования бизнес-процессов [695]. Этот подход обычно начинается с создания так называемой "детальной картины", или модели бизнес-процесса, составленной в свободном стиле и используемой в методе SOMA.

Можно отметить взаимосвязь между методом программных систем SSM (Soft Systems Method) Чекленда (Checkland) и методом SOMA. Знаменитая аббревиатура Чекленда CATWOE означает: Customers (пользователи), Actors (исполнители), Transformation processes (процессы трансформации), Weltanschauung (идеология), Owners (владельцы) и Environment (среда). Эти элементы в SSM определяют содержание компонентов модели системы и напрямую соответствуют концепциям SOMA (табл. 8.1). Кроме того, введенное Чеклендом понятие базового определения задачи системы напрямую соответствует понятию миссии.

Таблица 8.1. Общие понятия в методах SSM и SOMA

SSM	SOMA
Пользователи — жертвы или лица, получающие преимущества от T	Внешние агенты — заинтересованные лица, пользователи, спонсоры, инспекторы, внешние системы и т.д.
Исполнители — те, кто реализует T	Исполнители — внутренние агенты
Процесс преобразования (T) — преобразование входных данных в выходные	Система
Идеология — общественное мнение, обеспечивающее значимость T в общем контексте	Цели, задачи, метрики, допущения, исключения и т.д.
Владельцы — те, кто может прекратить T	Спонсоры
Ограничения среды — элементы внешней среды системы, на которые мы не можем влиять	Внешние объекты, таймеры, предположения и т.д.

В SOMA, как и в методе моделирования, не учитываются сделанные в [157] тонкие различия между системой, как моделью мира, и миром, как системой. Здесь точность не играет роли, так как речь идет о моделировании. С этими авторами можно согласиться по вопросу важности активной роли субъекта в познании, но нельзя согласиться с отсутствием объективного базиса процесса познания.

Объективно мир является системой, но система — это в некотором смысле субъективная идея. Однако в SOMA мы строим "модель мира" в форме моделей бизнес-процессов и задачи, а "модель системы" — в виде модели спецификации Catalysis. Здесь не место для исследования этих важных философских идей. На практике не существует проблемы в использовании SOMA как метода моделирования программных систем; его нечеткие расширения описаны в приложении А.

Процесс инженерии требований в SSM также подобен подходу в методе SOMA. В табл. 8.2 приводится семь стадий метода SSM и соответствующие эквиваленты из метода SOMA.

Таблица 8.2. Стадии SSM и SOMA

SSM	SOMA
Определение ситуации	Определение миссии, задач, составление модели бизнес-процессов
Описание ситуации (детальная картина)	Создание модели бизнес-процессов
Выбор точки зрения и введение базовых определений	Определение миссии и основных задач
Построение концептуальных моделей функций системы для каждого базового определения	Создание бизнес-модели и объектной модели задач
Сравнение концептуальной модели с миром	Построение объектной модели задач, модели бизнес-объектов, сквозной контроль (трассировка)
Определение возможных и желательных изменений	Определение задач, обсуждение на семинарах

В этом разделе дается только краткий и довольно поверхностный обзор подходов к инженерии требований (Requirements Engineering), так как основная цель состоит в том, чтобы представить собственный метод. Многие методы инженерии требований, кратко описанные в этом разделе, включены в SOMA, и я подтверждаю мою приверженность им. В моем подходе основное внимание уделяется доказательству того, что системная спецификация служит представлением требований. В SOMA для этого применяется математический формализм, но я решил не придерживаться его в этой книге, так как в действительности более важны практические проблемы, с которыми приходится сталкиваться изо дня в день.

8.2. Инженерия требований и системная спецификация

Наиболее общее неверное представление при обработке данных заключается в предположении, что требования клиентов совпадают со спецификацией системы, которая должна удовлетворять этим требованиям. При таком предположении можно прийти к заключению, что анализ прецедентов является единственным методом моделирования требований. Джексон [412] отрицательно отзывается об этой идее, указывая, что прецеденты полезны для определения систем, но не позволяют полностью описать требования. Прецеденты связывают исполнителей, которые представляют роли пользователей в системе, с самой системой. С другой стороны, требования могут выдвигать те люди и организации, которые никогда не окажутся вблизи системы.

На рис. 8.1 мы видим часть живописно проиллюстрированной интерпретации суждения Джексона. Документ определения требований должен быть написан на языке, принятом в области применения системы. Спецификация необходима только для описания интерфейса

системы и поэтому включает различные обозначения. Спецификация S описывает интерфейс явлений, общих для среды (мира) и системы. Для ее выражения могут быть применены прецеденты. Модель требований R — описание этих и других явлений в мире. R зависит и от спецификации, и от мира. Джексон также заявляет, что "клиента обычно интересуют результаты, которые заметны на некотором расстоянии от машины".

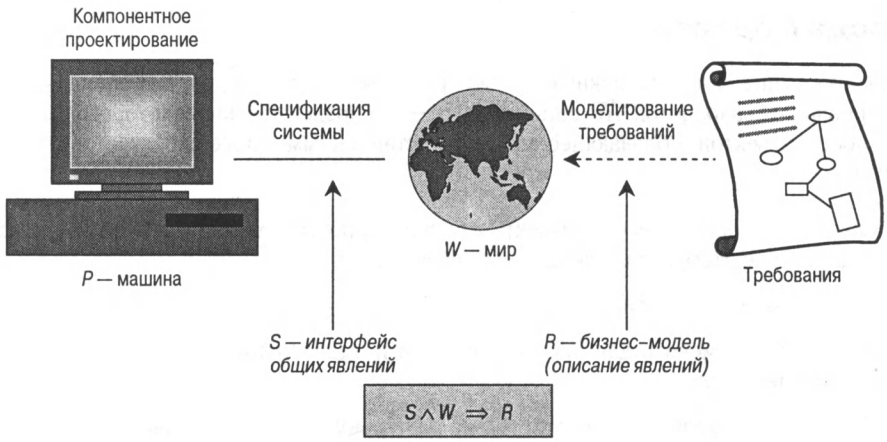


Рис. 8.1. Модель спецификации — это не модель требований

Игнорирование взаимодействия с непользователями может привести к потере важных технических возможностей. Я работал с системой, обрабатывающей заказы и автоматически оценивающей их стоимость. Назначение этой системы состояло в приеме заказов от клиентов и автоматической оценке их стоимости с использованием различных (часто комплексных) механизмов ценообразования на основе технологии брокеров объектных запросов. Проблема заключалась в том, что некоторые заказы были очень сложными или очень большими для автоматической обработки. Они рассматривались продавцом, который, конечно, имел связь с "системой". Все шло хорошо, на экране отображались "недопустимые" (или "обрабатываемые вручную") заказы. Для таких заказов продавец применял электронные таблицы и другие программы. Но появились дальнейшие проблемы — некоторые заказы были так сложны, что квалификации и опыта продавца в области финансовой математики не хватало для решения задачи. Для таких заказов продавец должен был переговорить со специалистом, имеющим ученую степень в области финансовой обработки. Так как мы использовали метод SOMA, то моделировали и этот вариант "нестандартного прецедента", и в результате, когда наш специалист увидел построенную модель, то немедленно реализовал ее в системе. Как следствие, мы смогли радикально улучшить бизнес-процесс и обслуживание клиентов. Даже это относительно малое отклонение от рамок системы привело к большим доходам. Во многих более сложных случаях важность выхода за пределы системы будет еще больше.

Моя интерпретация аргументов Джексона заключается в том, что нам необходим особый метод моделирования бизнес-процессов, совместимый с моделями прецедентов из спецификации. Альтернативой является возвращение к "матрешке" вложенных моделей, описанных в терминах "бизнес-прецедентов" [411]. Этот подход не только устарел, но и не в состоянии

решить упомянутые выше проблемы. Таким образом, прежде чем продолжить изложение, необходимо получить ответы на следующие два вопроса.

- Что такое модель?
- Что такое бизнес-процесс?

Природа моделей

Моделирование является важной частью программной инженерии, а особенно объектно-ориентированных разработок. **Модель** (model) — это представление некоторого объекта или совокупности объектов, обладающее всеми представленными ниже свойствами или некоторыми из них.

- Она всегда *отличается* от объекта или совокупности моделируемых объектов (оригинала) по масштабу, реализации или поведению.
- Она имеет форму оригинала.
- Ее можно применять для прогнозирования поведения или свойств оригинала (имитационная модель).
- Всегда имеется некоторое соответствие между моделью и оригиналом.

Примеры моделей в изобилии имеются в повседневной жизни: экспериментальная модель самолета в аэродинамической трубе, архитектурные модели в масштабе, электрические модели транспортных сетей, программные модели сгорания топлива в двигателях. Конечно, *любое* программное обеспечение является моделью некоторых явлений или процессов реального мира, так же как все математические уравнения есть аналитические модели.

Джексон [411] устанавливает связь между моделью и описанием, говоря, что моделирование предметной области включает создание набора обозначений для элементарных примитивов этой области, которые затем используются для описания свойств, взаимосвязей и поведения объектов в этой области. Например, если взять область отправления поздравительной открытки одному из друзей, то можно ввести следующие обозначения.

```
P - друг;
d - дата (день и месяц);
B(p, d) означает, что p родился в d.
```

Затем можно записать следующее.

```
Для каждого p существует только одно B.
```

Джексон считает, что обозначения, используемые при моделировании, должны в равной степени соответствовать и модели, и оригиналу. В контексте компьютерных моделей это может означать, что экземпляры класса `Friend` или записи баз данных должны соответствовать нашим реальным друзьям. Эта идея Джексона, названная М-концепцией, проиллюстрирована на рис. 8.2.

Объекты в предметной области и компьютере различны: в реальном мире друзья не находятся в секторах диска. Существует много объектов предметной области, которые не входят в

модель, например ноги наших друзей или их особые приметы. Многие аспекты компьютерной реализации не имеют отношения к модели, например разделение процессорного времени. Модель включает общие аспекты объектов из предметной области и компьютерной реализации.

Понимание сущности модели дает ключ к решению проблемы объектного моделирования. Мы должны ясно понимать, что так называемая модель бизнес-объектов описывает и состояние предметной области, и его возможную машинную реализацию. Для понимания и обоснования требований необходимо начать с модели предметной области.

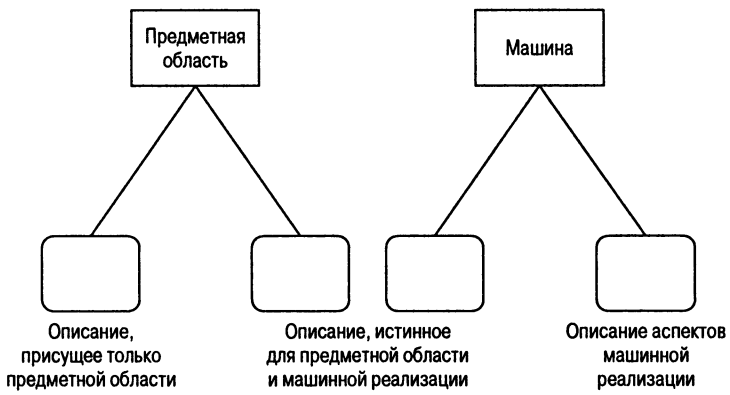


Рис. 8.2. Модель по Джексону

На мой взгляд, объектное моделирование является общим методом представления знаний. Мало что нельзя промоделировать посредством объектов, если, конечно, мы не ограничиваем себя семантикой некоторого конкретного объектно-ориентированного языка программирования. С помощью объектов можно моделировать и предметную область, и работу компьютера. Это не значит, что мир состоит из модельных объектов. Просто модель всегда может иметь объектное представление. Хороший пример предложен в [188], где делается строгое различие между *моделью сущностей* (т.е. предметной области) и *спецификацией*. В работе утверждается, что мир не состоит из объектов, поскольку солнце не будит птиц по утрам, передавая каждой сообщение. Однако для обеспечения соответствия между моделями предметной области и спецификацией событие восхода солнца можно поместить на "классной доске" для оповещения каждого класса птиц. На самом деле в реальном мире все происходит иначе, но это хорошая модель для широковещательной рассылки сообщений. Конечно, во многих случаях объектное моделирование не уместно, особенно это касается научных расчетов. Например, вряд ли решение дифференциальных уравнений стоит моделировать с помощью объектов. Джексон [411] выдвигает еще одну полезную идею. Он утверждает, что каждый может выделить шаблоны анализа (он их называет проблемными фреймами), для которых применимы соответствующие методы моделирования. Такой подход на основе классификации обеспечивает возможность решения типовых проблем. Например, для создания простых фреймов применим метод JSP, который можно рассматривать как разновидность объектно-ориентированного проектирования. В [460] предлагается решать эту задачу в рамках метода KISS. Проблемные фреймы Джексона можно критиковать за учет лишь нескольких известных ситуаций, которые не помогают найти решения новых проблем. При этом Джексон не указывает конкретного способа применения метода. Однако это же можно

сказать о любом подходе, ориентированном на шаблоны, а эффективность таких подходов широко признана.

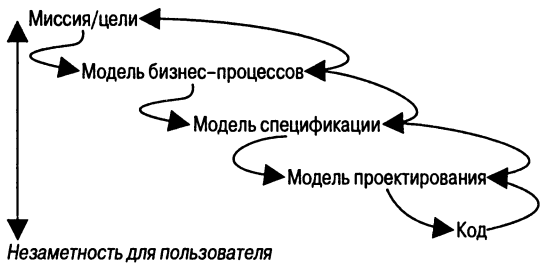


Рис. 8.3. Трассировка моделей

М-концепция Джексона (перенесение свойств одного предмета или явления на другой на основании признака, общего для обоих) обеспечивает понимание соответствия между двумя мирами (в приведенном выше примере — предметной областью и машинной реализацией). Однако чтобы применить объектное моделирование к реальной проблеме, от этого придется отказаться, как от слишком упрощенного представления. На практике приходится строить последовательность связанных друг с другом моделей. В [151] предлагается несколько более общее решение для поддержки этого представления. Там проводится различие между двумя методами объектного моделирования, которые характеризуются как конструктивистский (чистый) и импрессионистский (черновой). Черновой метод не отражает точную семантику объектного моделирования. Его сторонники считают, что "результаты анализа конкретизируются в процессе проектирования", но не определяют, что это означает. Понятие точности вводится только на этапе реализации, когда приходится сталкиваться с семантикой языка программирования. Чистый подход к моделированию существенно отличается. Он предполагает исследование вычислительной модели анализа. Считается, что модель анализа — это почти выполняемый код. Примеры близких по духу подходов описаны в [151, 188, 204]. Мы будем моделировать миссию и цели, взаимодействия между агентами, вовлеченными в бизнес-процессы, задачи, которые они выполняют, и бизнес-объекты. Как мы увидим, каждая из этих моделей преобразуется в следующую. При каждом преобразовании моделей проверяется их согласованность (рис. 8.3). Это гарантирует, что конечная система будет удовлетворять требованиям заинтересованных лиц. Как мы увидим, некоторые из этих моделей являются объектными, а некоторые — нет.

BPR

Перепроектирование бизнес-процессов (BPR — Business Process Reengineering) привлекает внимание специалистов с первой половины 1990-х годов. Однако изначально этот процесс не носил универсального характера, хотя и помог нескольким предпринимателям реорганизовать их деятельность, а также радикально и необратимо изменить отношение к информационным технологиям. Обычно этот процесс был связан с эволюционным развитием или адаптацией к объектной технологии и электронной торговле.

Перепроектирование бизнес-процессов — это радикальное переосмысление всей коммерческой деятельности, ее процессов, организационной структуры, управления, работ и стоимости системы. Цель BPR — получение разительных усовершенствований. Его реализация

начинается с определения ключевых процессов, необходимых для коммерческой деятельности, а затем переходит к сфере организационной структуры и функциональности, которые могут служить помехой такому процессу. Процессы рассматриваются под новым углом зрения с учетом организационных структур, созданных для их поддержки. С возрастанием глобальной конкуренции многие бизнесмены вынуждены реализовать BPR.

BPR выполняется в соответствии с бизнес-требованиями при полной поддержке главного менеджера, а также постоянной и открытой обратной связи с личным составом. Реорганизация бизнес-процессов обычно начинается с отдела IT, поскольку его деятельность связана со многими другими отделами.

В рамках BPR корпоративная информация предоставляется всем участникам бизнес-процесса, и каждый из них может высказать свое мнение и поучаствовать в принятии решения. К сожалению, это больше искусство, чем наука, потому что аналитик интуитивно находит то, что иногда называют "грандиозной идеей". Все приводимые в литературе примеры связаны с такой грандиозной идеей, например [354] и [217]. Компания Ford смогла уменьшить количество своих счетов с 500 до 125, оплачивая товары по получении и не занимаясь приведением в соответствие счетов и заказов. Компьютерная система справлялась с задачей, но грандиозная идея была сгенерирована исходя из того, что в компании Mazda аналогичный отдел состоял только из пяти человек. Аналогично компания Mutual Benefit Life сократила время обработки заказов с 24 часов до 4 за счет использования экспертной системы. Здесь грандиозная идея возникла из оценки технологических возможностей экспертных систем и размышлений над протеканием бизнес-процессов. Я думаю, что главный урок из этих и других примеров состоит в том, что необходим более систематический метод. Специалистам по перепроектированию необходимы средства представления и моделирования, позволяющие выполнять анализ на модели, средства проектирования бизнес-процессов и оценки проектных решений. Объектная технология обеспечивает решение этих задач, потому что бизнес-процессы можно моделировать с помощью связи объекта с его состояниями, обязанностями и, что более важно, знаниями о бизнес-политике и правилах. Еще один урок заключается в том, что иногда процесс BPR становится возможным только благодаря IT. Известна история об управляющем директоре страховой компании, который в 7 часов вечера (когда других сотрудников уже не было в офисе) смог решить проблемы клиентов, так как имел доступ к модели работы компании через терминал и мог обращаться к счетам клиентов.

IT помогает устранить ненужные потоки, превращая менеджеров в клерков, а клерков — в лиц, принимающих решения. Это сводит на нет классическую идею о разделении труда Адама Смита (Adam Smith) и Фредерика Тейлора (Frederick Taylor). В самом деле, высказывания экспертов современной науки об управлении очень напоминают прогнозы Ленина, сделанные в работе *Государство и революция* [484][1], в которой он утверждал, что в истинно свободном и высоко производительном обществе разделение труда должно быть окончательно уничтожено. Он предлагал управлять процессами, а не людьми. Это очень похоже на идеи Тома Петерса [625], призывающего наделить правами рабочую силу и исключить многоступенчатость принятия решений. Однако в результате сопоставления этих различных политических взглядов неизбежно напрашивается вопрос: *допустят ли такую радикальную перестройку политико-экономические структуры и поддержат ли такой способ организации труда.* Петерс, очевидно, верит в это, но не доказано и, вообще, маловероятно, что такой положительный пример получит развитие на всех предприятиях мира и приведет к соответствующим

[1] Ленину были очень близки идеи Тейлора. Он даже сформулировал лозунг "Электрификация+ Тейлоризм=Социализм".

социальным изменениям. Еще одна, более насущная социальная проблема заключается в том, что менеджеры должны контролировать работу рабочего коллектива. Такой надзор может вызвать чувство обиды. В [527] высказывается мысль о том, что если менеджеры считают такой контроль полезным, то они и сами должны ему подчиняться.

В литературе по BPR подчеркивается, что построение функциональных моделей не допускает последующих изменений из-за чрезмерной специализации. Это не означает, что объектные модели лучше, но существует общее мнение, что элементом анализа должен быть бизнес-процесс. Роль объектов заключается в обеспечении прямой поддержки моделируемого процесса посредством взаимосвязей между исполнителями. Я сказал бы, что любой естественный язык описания бизнес-процесса естественно переносится на объектную модель. Объектно-ориентированное описание можно применять для имитационного моделирования бизнес-процессов, которые, в свою очередь, могут быть использованы для сценарного анализа и поиска "грандиозной идеи".

В BPR основное внимание уделяется процессу, а не функциональной специализации. Ориентация на процесс приводит к следующим преимуществам:

- повышение производительности и эффективности;
- сокращение стоимости;
- увеличение гибкости и адаптируемости процесса;
- большее удовлетворение от работы;
- повышение качества товаров.

Для контроля реализации этих преимуществ назначается "инженер по организации производства". Уроки BPR могут быть применены непосредственно к процессу разработки объектно-ориентированного программного обеспечения, где роль инженера по организации процесса будет в значительной степени определяться специалистом по анализу предметной области.

Проекты BPR не достигают успеха по многим причинам. Отметим некоторые из них.

- Невозможность использовать средства, выделяемые на развитие информационных технологий, а также для управления существующими структурами и бизнес-процессами.
- Чрезмерные надежды на пакеты программного обеспечения и ERP-системы и уверенность в том, что все проблемы решаются путем нажатия клавиш.
- Отсутствие ясной стратегии и противодействие со стороны управляющих структур.
- Отказ от радикального сокращения управленческих структур.
- Препятствия со стороны управляющего персонала.
- Чрезмерный упор на сокращение стоимости.
- Недостаток терпения — бюджет урезается прежде, чем появятся ощутимые результаты.
- Недостаток понимания со стороны клиентов.

Наряду с неудачами, имеются и широко известные успехи в таких компаниях, как Baxter Healthcare, IBM, British Telecom, Lucas, Nabisco и Xerox. Строительная компания National & Provincial Building Society была реорганизована в 90–95-х годах и в последний год сообщила об увеличении прибыли и доходов на 45% без значительного сокращения основных фондов.

Зачастую информационные технологии "привязывают" организацию к существующим методам работы. Это происходит, в основном, потому, что изменения в компьютерных системах требуют слишком много времени. Таким образом, в рамках BPR очень важно быстро адаптировать IT. Для этого необходима гибкая среда программирования, позволяющая быстро построить и легко развивать прототипы. Повторное использование библиотек позволяет сделать разработку быстрее. Иными словами, объектно-ориентированные методы должны обеспечивать возможность гибкой настройки модели во время ее работы.

Конечно, разработка нового бизнес-процесса не может быть выполнена без моделирования, и объектная метафора идеально подходит для моделирования. Эволюционная разработка также почти всегда представляет возможность заново проектировать некоторые аспекты бизнес-процессов. Эта глава показывает, как объектно-ориентированные методы моделирования систем могут быть объединены с моделированием бизнес-процессов и разработкой технических требований. Мы увидим, как применять язык объектного моделирования, рассмотренный в главе 6, к моделированию бизнес-процессов и задач пользователей. В следующих разделах будет показано, как, начав с объектной модели бизнес-процесса, плавно перейти к объектной модели задач агентов, которые имеют непосредственное отношение к целям и задачам предприятия. Это, в свою очередь, приведет к модели прецедентов UML. В заключение будет показано, как из этих моделей получить обычную объектную модель. При этом модель прецедентов может плавно перейти в спецификацию компьютерной системы.

8.2.1. Совместная работа, автоматизация технологического процесса и программное обеспечение коллективного использования

За последние 50 лет производительность труда в производстве сильно возросла. В то же время производительность труда офисных работников не увеличилась, несмотря на увеличение применения компьютеров [126, 662]. Согласно данным ассоциации управления информацией и других исследователей, от 94 до 95% офисной информации все еще содержится на бумаге и на ее поиски затрачивается от 15 до 30% офисного рабочего времени.

В этом разделе показано, как IT позволяет перестроить бизнес-процессы и повысить производительность труда офисных работников. Это можно реализовать по-разному: за счет компьютерной поддержки совместной работы (CSCW — Computer Supported Cooperative Work), автоматизации последовательности выполняемых действий и программного обеспечения для коллективной работы. Этой цели служат различные технологии, такие как управление изображением документа, хранение информации на CD-дисках, широковещательная рассылка данных по сети, базы данных, способные к самовосстановлению, и специальные технологии, основанные на концепциях лингвистики, искусственного интеллекта, биологии, антропологии и других общественных наук.

Системы управления технологическими процессами автоматизируют существующие повторяемые процессы, в то время как программное обеспечение коллективного использования поддерживает *специальные* формы сотрудничества и координирования. Некоторые

программные продукты поддерживают оба метода. Приведенное разбиение отражает разные позиции специалистов по реорганизации бизнес-процессов, которые сводятся либо к автоматизации и модернизации существующих процессов, либо к полному вытеснению существующих методов работы. Программное обеспечение коллективного использования успешно применяется в небольших организационных структурах, в которых принят групповой метод работы. Зачастую автоматизированные системы управления технологическими процессами и групповое программное обеспечение используются в рамках корпоративной сети предприятия.

Чтобы понять, когда лучше применять групповое программное обеспечение, обратимся к рис. 8.4. На этом рисунке уровень общественного взаимодействия, необходимый для выполнения задачи, сопоставляется с уровнем управления, к которому относится задача. В ячейках таблицы приводятся некоторые типичные приложения. Из рисунка видно, что программное обеспечение коллективного использования может применяться при высокой степени взаимодействия для всех уровней управления. При средней степени взаимодействия членов коллектива групповое программное обеспечение тяготеет к операционному уровню. Низкому уровню взаимодействия более соответствует автоматизация последовательности выполняемых действий. Если ожидаются организационные изменения, благоразумно перейти на гибкие системы, чтобы охватить оба типа приложений.

Уровень взаимодействия	Уровень управления		
	Оперативный	Средний	Верхний
Низкий	Доступ к базам данных	Генерация отчетов	Производственные информационные системы
Средний	Электронная почта, совместное использование ресурсов	Расписание встреч	Системы поддержки принятия решений, экспертные системы, системы управления складом
Высокий	Неформальные обсуждения, встречи	Системы поддержки принятия решений, экспертные системы, системы управления складом	Групповое принятие решений

Рис. 8.4. Области применения программного обеспечения коллективного использования и автоматизации последовательности выполняемых действий

Инструментальные средства для автоматизации последовательности действий распадаются на три основных класса: системы управления изображением документа (DIM — Document Image Management), аналогичные системе Filenet с ее неотъемлемой частью — языком Workflo; системы автоматизации офисной работы; и специфические продукты группового использования. Такие системы высокоэффективны в бюрократических подразделениях, таких как отделы местного самоуправления, где бумажный поток и строгое следование законам есть важным фактором официальной отчетности. В этих случаях количество хранимых бумаг и сопутствующие затраты могут быть существенно сокращены. Продукты, подобные Lotus Notes,

и различные корпоративные системы дают возможность просматривать поступающие документы и сохранять их, используя электронную почту. Это помогает избежать больших затрат при рассылке копий документов людям, которые не интересуются присланной информацией, в то время как искренне заинтересованные стороны остаются без внимания. Многие продукты предлагаются в двух версиях: для разработчиков и для пользователей. Версии для разработчиков обычно включают языки сценариев высокого уровня. В некоторых продуктах, таких как Beyond Mail, применяются правила маршрутизации почты для поддержки автоматизации последовательности выполняемых действий. Существует возможность хранить и передавать стандартные ответы по электронной почте. Необходимость такого подхода не должна быть ограничена локальной вычислительной сетью. Многие системы электронной почты поддерживают протоколы X400 и X500.

В HCI-проектировании основное внимание уделяется детальному выполнению конкретных задач. Такой подход хорош при анализе использования компьютерной системы администратором рабочей станции, но не годится для автоматизации последовательности действий, в которой участвует много пользователей. Результаты HCI-проектирования можно расширить до технологий CSCW. При совместной работе на первый план выступают социальные аспекты, поэтому при ее автоматизации следует учитывать результаты общественных наук, в том числе антропологии, этнометодологии и общественной психологии. Этнометодология, как мы видели в разделе 8.1, изучает отношения между людьми (персоналом), группами, а также модели и технологии совместной работы. Она широко применяется при изучении воздействия IT на отдельных людей и организации в целом. По сути, технология CSCW больше приспособлена для совместной работы, чем HCI [227]. Специалисты по этнометодологии изучают людей, совместно работающих в группах, и регистрируют физические перемещения, жестикуляцию, фокус глаз и их отношение к диалогу. Анализ диалога дает удивительные результаты. Например, как оказывается, звуковые сигналы очень важны: пользователю легче завершить переговоры через систему мультимедиа, если в конце беседы слышен звук закрывающейся двери. Участники видеоконференций менее чувствительны к жестикуляции собеседника, чем при диалогах лицом к лицу. По этой причине некоторые пользователи пытаются усиленно жестикулировать. Однако поведение в контексте видео может выглядеть очень глупо, если рассматривать его в офисном варианте.

Очень часто при автоматизации последовательности выполняемых действий и разработке программного обеспечения коллективного использования важную роль играет анализ прецедентов и, в более общем смысле, взаимодействия агентов. Каждый сценарий взаимодействия связан с набором объектов, в том числе с отчетами баз данных и документами. Анализ сценария ведет к объектно-ориентированному описанию, и эта модель может быть использована для критического анализа процесса автоматизации. Представление процессов объединяет в себе результаты объектно-ориентированного анализа и методов проектирования, в том числе метода Ptech (см. приложение B).

Было бы ошибкой считать, что выгоды от совместной компьютерной работы достигаются без дополнительных затрат. Кроме затрат на новое программное обеспечение, необходимы затраты на обучение и на изменение управления. Смею утверждать, что такой стиль работы требует, чтобы каждый член коллектива имел собственную рабочую станцию. Если их нет, необходимо приобрести. С другой стороны, существуют также некоторые скрытые выгоды. Прямой доступ к информации может ускорить время "выхода на рынок". Например, первоклассная автоматизированная система может сократить число встреч, требуемых для решения критических вопросов, ибо каждый может свободно собрать недостающую информацию и выставить встречные требования в процессе повседневной работы.

8.3. Декомпозиция больших задач

Возвращаемся к главной идее изложения — проектированию объектной модели бизнес-процессов. Для небольшого бизнеса это просто, но, к сожалению, при построении объектной модели большой корпорации, такой как AT&T, British Petroleum, Chase Manhattan или IBM, это так сложно, что, в сущности, не имеет смысла. Осуществление этого замысла может занять слишком много времени. Подходы, основанные на построении детальной картины или ORCA, не помогут в решении проблем такого масштаба. Что же можно сделать, чтобы охватить такие масштабные проблемы, где объектное моделирование уместно и эффективно? Чтобы ответить на этот вопрос, я предлагаю метод Mission Grid (анализ миссии).

При анализе деятельности любого коммерческого предприятия сначала необходимо выяснить, кто является его клиентами, а кто относится к заинтересованным лицам. Обычно в число заинтересованных лиц включают (в дополнение к вездесущим клиентам) поставщиков, регулирующие органы, торговые ассоциации, информационных поставщиков и конкурентов. Как только этот круг определен, можно выделить внешние цели. В литературе они называются по-разному: совместными целями или предложениями для пользователей (CVP — Customer Value Proposition) [418]. Эти цели отражают желания предприятия и его клиентов, например поставка продуктов высокого качества по приемлемой цене. Это подходит клиенту, которому необходим данный продукт, и он желает приобрести его у надежного поставщика, который остается на рынке и продолжает улучшать свой товар. Метод анализа миссии предполагает описание этих задач вдоль левого края электронной таблицы. Внешних задач недостаточно, чтобы охарактеризовать цели предприятия, поэтому мы должны также определить внутренние цели. Внутренние цели — это те, которые организация ставит перед собой. Например, задача соблюдения законодательства приводит к необходимости регистрировать счета и налоговые декларации. Для ясности они записываются в правой части электронной таблицы (рис. 8.5).

Теперь необходимо заполнить таблицу (описать процессы) таким образом, чтобы обеспечить выполнение внешних и внутренних задач. Затем эти процессы должны быть перегруппированы таким образом, чтобы можно было определить "логические роли". Логическая роль — это абстрактное описание работ, для выполнения которых требуются общие навыки (рис. 8.5). Представление, показанное на рис. 8.5, можно назвать сеткой заданий.

Зачастую разрабатываются две сетки заданий: первая представляет текущую ситуацию, а вторая — видение заново проектируемого бизнеса. В идеале они разрабатываются независимо.

Сетка заданий является превосходным инструментом для построения общей стратегии предприятия. При ее построении всегда нужно оценивать, является ли данная цель необходимой и устойчивой в конъюнктурных условиях рынка.

При построении сетки следует оценивать, предполагает ли процесс интерфейс с пользователем и является ли дифференцирующим.

Термин "интерфейс с пользователем" означает, что процесс подразумевает прямой контакт с клиентом. Обычно неблагоразумно доверять такие процессы третьим лицам. Многие компании, в недавнем прошлом доверявшие его "справочным столам" (системам поддержки пользователей в сети), начинают понимать свою ошибку. Дифференцирующими называют те действия, которые характеризуют компанию в глазах ее клиентов. Например, если в рекламном объявлении организация гарантирует, что будет брать определенный процент с вашего дохода каждый год, и при этом не обеспечивает никакого обслуживания, то можно быть твердо уверенным, что это рекламное объявление относится к налоговой инспекции. Оно выделяет

(дифференцирует) эту организацию среди всех других. Дифференцирующие процессы тоже нельзя доверять третьим лицам. Таким образом, сетка заданий является мощным инструментом для пересмотра проектируемого бизнес-процесса. Она также является отправной точкой для разработки систем управления предприятием и объектно-ориентированного моделирования предметной области.

	Повар	Шеф-повар	Уборщица	Бухгалтер	
Представить блюдо	Приготовить блюдо	Сходить на рынок			
	Нарезать	Описать рецепт	Убрать рабочее место		
Проинформировать посетителей	Получить совет	Научить повара			
		Организовать рекламу			
				Выполнить финансовый анализ	Оценить прибыль
	Получить оплату		Содержать руки в чистоте	Вести бухгалтерские расчеты	Вести бухгалтерию

Рис. 8.5. Фрагмент сетки заданий для ресторана

8.4. Исследование бизнес-целей и приоритетов

После построения сетки заданий можно определить специфические задачи процесса. В идеале для этого можно собрать объединенный семинар и пригласить на него всех заинтересованных лиц. Цели можно выписать на плакатах или представить на другом носителе информации (например, текст может выводиться с компьютера). Опыт показывает, что обычно формулируют около 13 целей. Это вызвано тем, что у людей иссякают идеи после долгого обсуждения, а 13 целей удобно расположить на двух страницах.

В методе SOMA никакой вид деятельности не обходится без тестирования. Этот принцип применяется и к определению целей. Например, если речь идет об управлении отелями и целью является обеспечение высокого качества обслуживания, то мерой качества может служить количество "звездочек". Конечно, в некоторых случаях точную метрику определить сложно. Метрики — важный инструмент для освещения, прояснения и достижения целей. Обсуждение метрик помогает более ясно представить цели и часто приводит к выявлению дополнительных целей или к модификации тех, которые уже зафиксированы. Большие затраты времени для обсуждения метрик — это не просто потеря времени.

Все цели необходимо расположить по приоритетам. Формально приоритеты целей можно определить из сетки заданий. На семинаре это занимает слишком много времени. Один из путей быстро расставить приоритеты — голосование участников семинара по каждой цели. Обычно каждому участнику делегируется определенное число голосов, соответствующее приблизительно 66% от числа целей (например, 9 голосов за 13 целей). При голосовании можно использовать цветные бумажные кружочки, которые продаются в отделах канцелярских товаров. Затем объясняются правила голосования: "Вы можете отдать все ваши голоса за одну цель или распределить их по нескольким (равномерно или неравномерно) согласно вашему мнению о важности цели. Нет необходимости использовать все голоса, но нельзя отдавать (или продавать) неиспользованные голоса другим участникам". После этого все должны подойти к плакату и расставить свои приоритеты. Это позволяет ввести динамику в процесс обсуждения.

Иногда необходимо провести два раунда голосования (две интерпретации). Тогда потребуются кружочки двух цветов, и результаты голосования объединяются. Рассмотрим пример двух возможных интерпретаций, которые могут быть объединены.

1. Голосование с вашей точки зрения как индивидуального пользователя.
2. Голосование с общей точки зрения.

или

1. Голосование с точки зрения поставщика.
2. Голосование с точки зрения клиента.

Результаты голосования часто порождают дальнейшие полезные обсуждения. Это позволяет заново расставить приоритеты. Такая ситуация зачастую возникает из-за перекрытия целей, которые выдвигаются на первый план.

Цель, которой не может быть присвоен приоритет, отбрасывается или переформулируется. Приоритеты — ключевой инструмент для управления проектом, так как они определяют, что должно быть (с точки зрения спонсора) выполнено первым. Конечно, следует учитывать технические возможности. Проблемы, которые не могут быть разрешены, регистрируются с указанием сотрудников, ответственных за их решение. Должны быть также зарегистрированы специфические предложения и исключения.

Теперь мы имеем общую сетку заданий для целой организации, состоящую из сложных процессов. Каждое задание связано с рядом небольших бизнес-целей, расположенных по приоритетам. Теперь можно приступать к построению модели предметной области и ее процессов.

8.5. Агенты, диалоги и бизнес-процессы

Как только цели расставлены согласно назначенным приоритетам, можно строить первую объектную модель — модель предметной области. Для этого нужно понять, что такое бизнес-процесс. Многие специалисты в области инструментальных средств и методов моделирования бизнес-процессов находят, что очень трудно ответить на вопрос: *что является бизнес-процессом*. Обычно они говорят, что бизнес-процесс представляет собой набор процессов, объединенных потоками данных, распределением времени для каждого процесса и возможным

объектно-ориентированным [453]. Типичными программными продуктами для такого моделирования и управления потоками данных являются BeyondMail от компании Beyond Inc., Epic/Workflow от Computron, TeamRoute от DEC, WaveFlow от HP, Folder Application Facility от IBM, ProcessWise от ICL, ProcessIT от NCR, Imageflow от Plexus, а также Filenet и Staffware. Многие из этих инструментальных средств построены на объектной технологии, даже несмотря на то что они не рассматривают процесс с точки зрения объектной ориентации. Например, утверждается, что в ProcessWise объектно-ориентированные методы используются для моделирования бизнес-процессов и разработки компьютерных систем, необходимых для их поддержки. Модуль интеграции ProcessWise Integrator позволяет разрабатывать системы управления потоками данных. Этот метод подобен в некоторых отношениях методу SOMA для моделирования систем и предприятий, но с акцентом на потоки данных. Это также иллюстрирует близкую связь между программным обеспечением коллективного пользования и реорганизацией бизнес-процессов. Все эти методы имеют общее свойство — отсутствие адекватной *теории* бизнес-процессов.

В отличие от многих методов реорганизации бизнес-процессов и разработки требований, SOMA предлагает строго определенный, теоретически основанный и перспективный подход к этой проблеме. Он основывается на науке семиотике и работе [800]. При этом прецеденты не рассматриваются как отправная точка проектирования, а выделяются из модели процесса.

Как я уже говорил, ключевой частью подхода SOMA к разработке систем является использование общих семинаров пользователей и разработчиков для формулировки требований и анализа. Эта методика появилась раньше объектной технологии, и я использовал ее задолго до возникновения объектно-ориентированных методов анализа. В процессе таких семинаров я пришел к выводу, что работу с пользователями нужно начинать не с методов статического моделирования. Получение модели предприятия или бизнес-процесса занимает довольно много времени. При этом разработчики пытаются навязать свою точку зрения на семинаре, так как они имеют больший опыт в таком моделировании. Довольно часто это ведет к построению статической модели. Многие специалисты пришли к заключению, что пользователей лучше всего привлекать к моделированию на основе процессов. Мой опыт показывает, что при этом нужно основываться на модели требований в бизнес-процессах и использовать строго объектно-ориентированную методику моделирования, которая является обобщением метода потоков данных и анализа прецедентов.

8.5.1. Модели бизнес-процессов

Инженерия требований и реорганизация бизнес-процессов должны начинаться с модели связей и соглашений между участниками коммерческой деятельности, другими заинтересованными лицами, клиентами, поставщиками и т.д. Тогда эти "внешние агенты" будут определены как часть сетки заданий.

Рассмотрим некоторую коммерческую деятельность или предприятие. Это может быть небольшая компания, отделение или отдел большой компании либо даже просто один торговец. **Бизнес-процесс** (или область торгово-промышленной деятельности) представляет собой сеть взаимосвязанных агентов. В [275] она называется сетью обязательств. **Агент** — это любое предприятие в мире, которое может обмениваться информацией с другими агентами. Это может быть клиент, управляющий, служащий, организационная единица, компьютерная система или даже механическое устройство некоторого типа, такое, например, как часы. Агенты автономны и гибки. Они отвечают за организацию стимулирующего воздействия и могут демонстрировать социальные аспекты, т.е. взаимосвязь. Обычно агенты обладают определенным уровнем

интеллекта. Это очевидно, когда речь идет о человеке, но когда речь идет о механических агентах, имеется в виду степень их "коммуникабельности". Напрашивается вопрос: *что же означает взаимосвязь двух агентов*.

Агенты могут быть **внутренними** по отношению к бизнес-процессу, который мы исследуем, или **внешними** по отношению к нему. И не имеет никакого значения, являются ли они пользователями системы, т.е. действующими субъектами (исполнителями). Агенты, подобно исполнителям, выполняют определенные роли.

Участники бизнес-процесса должны взаимодействовать с внешним миром и для этого использовать некоторые соглашения об обозначениях и сигналах. Эти сигналы между агентами можно назвать **семиотическими действиями**, которые *опираются* на некоторое материальное основание (субстрат). Они включают ряд семиотических уровней потоков данных с неявно выраженными общественными отношениями.[2] Например, основным средством взаимосвязи (основанием) могут быть заполненные формы офисных документов, а социальный контекст может состоять в том, что каждый принимает правила игры. Если основным средством взаимосвязи является естественный устный (или письменный) язык, то можно говорить о **речевых действиях**, или **диалогах** [46, 692]. В [275] доказано, что бизнес-диалоги имеют постоянную периодическую структуру, основанную на пяти примитивных действиях: утверждении, оценке, объявлении, предложении/обещании и запросе.

Семиотические действия (или диалоги, как я буду называть их в дальнейшем) могут быть представлены сообщениями, направленными от инициатора связи (источника) к получателю (цели). Одно и то же семиотическое действие может быть представлено различными видами сообщений[3]. Такой подход приводит к определению класса эквивалентных сообщений, и фактическое сообщение можно рассматривать как представитель класса. Несколько соглашений (контрактов) могут выражать одну и ту же взаимосвязь и представлять класс эквивалентных сообщений.

Типичный диалог представлен на рис. 8.6, где внешний агент потребителя размещает заказ, связанный с некоторым видом коммерческой деятельности. Это сообщение предполагает определенную форму ответа: "заказ принят" или "нет в наличии". Я думаю, что мы имеем совершенно законное право представить диалог с использованием обозначений диаграммы прецедентов UML (рис. 8.6, *а*). Если необходимо различать исполнителей (выделить потребителей), можно использовать обозначения, приведенные на рис. 8.6, *б*.

Сообщение представляет собой поток данных, значит, этот подход обобщает метод моделирования потоков данных, существенно улучшая его, поскольку данные передаются в обоих направлениях (запрос и отклик). Вот почему для обозначения конца линии связи выбран затемненный круг, а не стрелка. При этом линия направлена от *инициатора* связи, а не от источника данных.

Мы начинаем понимать, что агенты можно моделировать как объекты, передающие сообщения друг другу. Ясно также, что агентов можно классифицировать по различным типам. Следовательно, нашу модель бизнес-процесса можно назвать *агентной объектной моделью* (AOM). Возникает вопрос, как анализировать сообщения. Насколько мне известно, кроме метода ORCA, который в чем-то совпадает с моим методом, есть только три возможных приема.

---

[2] Семиотика — сравнительное изучение систем обозначений, которое имеет большое значение в таких важных областях, как математическая логика, создание естественного языка, антропология и литературная критика. Система обозначений может быть проанализирована, по крайней мере, на трех уровнях: синтаксиса, семантики и прагматики. Можно рассмотреть пять уровней, включая уровень, определенный общественными соотношениями производства.

[3] В качестве простого примера можно рассматривать один и тот же диалог, представленный сообщениями на английском, китайском, немецком языках или языке урду.

1. Можно использовать метод Джексона разработки систем JSD (Jackson System Development) [410], его объектно-ориентированный вариант или метод KISS [460]. На мой взгляд, такой подход больше подходит для проектирования систем, чем для стадии инженерии требований.
2. Метод прецедентов [417] — возможно, наиболее широко применяемый альтернативный подход, рекомендуемый разработчиками языка UML. Однако, как мы видели, этот метод годится, в основном, для разработки системных спецификаций.
3. Из этого следует, что предложенный здесь метод, основанный на семиотике, является наиболее доступным и реалистичным.

Семиотическое (или речевое) действие характеризуется семантическим и прагматическим (возможно, неявным) уровнями соглашения или контракта, понятными обеим сторонам. Прагматика контракта представляет общественные отношения, а сообщение — семиотическое действие.

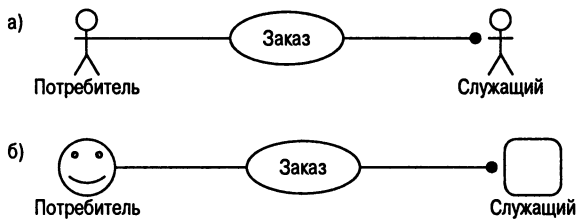


Рис. 8.6. Диалог

Мы считаем, что бизнес-процесс — это сеть взаимосвязанных диалогов между агентами, представленных посредством сообщений. В большинстве случаев это означает, что в результате взаимосвязи инициатор сообщения желает изменить состояние мира. Это желательное состояние мира является целью диалога, поскольку каждый диалог имеет цель или выходное условие, даже если это не заявлено (контракт, представляющий условия завершения диалога).

Цель достигается путем выполнения **задачи**. Здесь прослеживается два новых аспекта. Во-первых, выполняемые задачи могут быть сведены к нескольким стереотипам за счет выделения типичных задач, выступающих в качестве шаблона, по которому оцениваются реальные задачи и благодаря которым эти реальные задачи (или прецеденты) могут быть поставлены. Мне часто приходилось сталкиваться с возражениями о том, что при этом возрастает число прецедентов. Мой опыт показал, что это не приводит к существенному росту числа задач. Например, в простой модели обмена валют имеется только одиннадцать задач, из которых восемь очень малы (определены и проиллюстрированы ниже в этой главе). Во-вторых, как мы увидим в следующем разделе, задачи могут быть промоделированы как элементы объектной модели предметной области.

В модели бизнес-процессов возможны только серьезные целенаправленные диалоги, и поэтому можно утверждать, что каждый диалог состоит из шести компонентов и имеет следующую структуру.

1. **Инициирующее событие**, событие в мире, которое вызывает взаимодействие.
2. **Цель**, состояние мира, которого хочет достичь инициатор диалога.
3. **Предложение** или **запрос**, который содержит данные, необходимые для получателя.
4. **Переговоры**, посредством которых получатель определяет, совместимы ли цели и условия, что ведет либо к согласованию **контракта**, либо к отказу от предложения.

Контракт формализует цель и обеспечивает формальные условия для определения, достигнута ли цель.

5. **Задача**, которая должна быть выполнена получателем запроса для достижения цели и удовлетворения контракта. Это то, что обычно принимается за прецедент, когда один из агентов — исполнитель.
6. **Передача** результата выполнения задачи и любых связанных с этим данных, которые подтверждают, что условия выполнены.

Эту структуру обычно называют *диалогом для действия* [275, 800], но я добавил в нее понятие инициирующего события. Примечательно также, что существует симметрия предложений и запросов, поэтому каждое предложение можно заменить эквивалентным запросом, меняя исполнителя с получателем. В методе SOMA всегда приходится иметь дело с сообщениями в виде **запроса в канонической форме** (request canonical form). Флорес (Flores) представил теорию, проиллюстрированную на рис. 8.7, в терминах клиента (в нашей терминологии инициатор) и исполнителя (получатель), выполняющих простейшие речевые действия. Далее курсивом выделены действия в порядке их выполнения. Потребитель *заявляет* о своем участии и *определяет* требования к исполнителю (исполнитель *делает предложение* другому лицу, передавая сообщение о желании заключить с ним контракт). Затем следует процесс переговоров, целью которого является определение контракта, обязательного для исполнителя и принятого клиентом. Эти и другие этапы диалога могут включать рекурсию, посредством которой выполняются дополнительные диалоги. По завершении переговоров в контракте определяются условия, при которых достигаются цели клиента. Наконец, результаты этой работы *объявляются* завершенными и передаются клиенту, который должен *выразить* удовлетворение.

На рис. 8.8 показана структура диалога в методе SOMA, включающая рекурсию, которая может происходить на каждом этапе диалога.

Рассмотрим конкретный пример покупки дома. Инициатор может спросить: "Вы хотите купить мой дом?", а покупателю необходимо решить, нужно ли вести переговоры относительно цены. Эти переговоры могли бы включать (рекурсивно) вспомогательные диалоги между покупателем и ипотечным поставщиком, а также специалистом по строительству. Если все согласовано, то контракт будет принят и подписан (в буквальном смысле). Теперь начинается следующий этап выполнения задачи; в Англии он называется *составление актов* передачи прав собственности на недвижимость. Выполнение задачи включает в себя, среди прочего, поиск отчетов местных органов власти, документов на земельную собственность и т.д. Эти задачи выполняются по стандартным сценариям или блок-схемам, описанным в книге актов передачи прав собственности на недвижимость. Наконец, когда эта задача удовлетворительно решена, можно передавать ключи, и контракт считается *завершенным*.

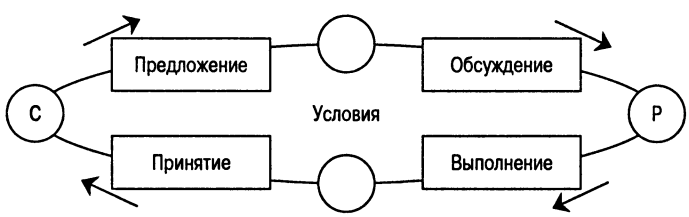


Рис. 8.7. Диалог по выполнению действия

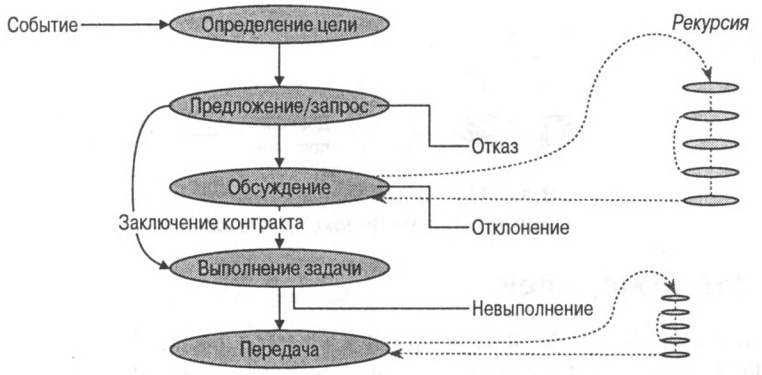


Рис. 8.8. Структура диалога в методе SOMA

Пример

Конечно, при реорганизации бизнес-процесса мы стремимся охватить не только сообщения, пересекающие его рамки, например размещение заказов, но и модели связей между клиентами, поставщиками, конкурентами и т.д. Это обеспечивает возможность предложения новых услуг, возможно, заимствуя их внутренние действия.

На рис. 8.9 и 8.10 показано, как эту методологию можно применить в простом случае поставки медикаментов, основываясь на примере больницы Baxter Healthcare [710]. Первоначально заказ от больничного отдела снабжения передавался на склад. Больница отвечала за хранение товаров и доставку в соответствующую операционную (рис. 8.9). После реорганизации товары, в том числе хирургические перчатки, доставляются поставщиком прямо в операционную, в которой они требуются (рис. 8.10). Конечно, сообщение `Order-Gloves` при этом изменилось. Это дает определенные преимущества, уменьшает количество больничного инвентаря и общую стоимость снабжения. Это, правда, делает больницу более зависимой от поставщика.

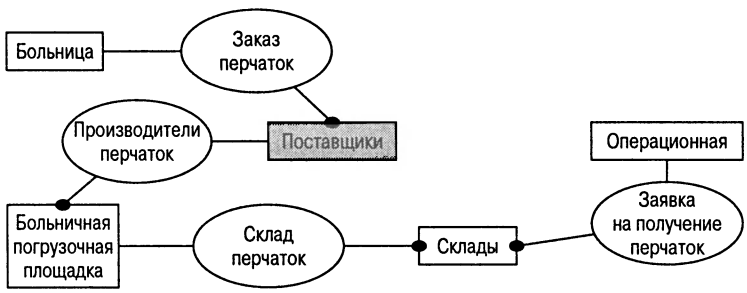


Рис. 8.9. До реорганизации системы материально-технического снабжения

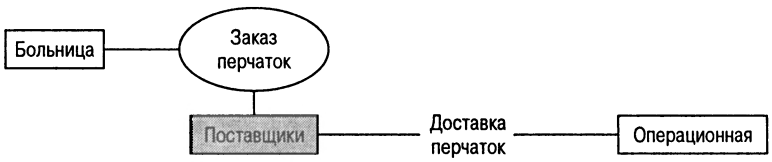


Рис. 8.10. После реорганизации системы материально-технического снабжения

Соответствие моделей

Проанализировав бизнес-процесс в терминах диалогов, сосредоточимся на этапе выполнения задачи диалога. Если в выполнении задачи участвуют исполнители, то этот этап называют прецедентом. Сначала напомним порядок построения модели. Мы начинали с определения миссии, на основе которой строили несколько процессов. Каждый процесс имеет несколько целей. Так получалась сеть диалогов (сообщений). Здесь необходимо задать себе два чрезвычайно важных вопроса.

- Поддерживает ли каждое сообщение выполнение *хотя бы одной* цели?
- Поддерживается ли *каждая* цель хотя бы одним сообщением?

Если ответ на каждый вопрос — *нет*, то в модель должны быть внесены поправки, поскольку мы либо упустили некоторые диалоги, либо моделируем диалоги, которые не способствуют достижению цели. Конечно, мы могли пропустить важную цель, и в этом случае необходимо проконсультироваться с пользователями о необходимости изменения перечня целей. В противном случае мы имеем полное право на перепроектирование: нужно остановить работу, которая не ведет к цели.

8.5.2. Диаграммы видов деятельности и моделирование бизнес-процессов

UML обеспечивает только один вид обозначений для моделирования бизнес-процессов — диаграмму видов деятельности. Эта диаграмма является частным случаем диаграммы состояний, которая предназначена для описания состояния дел или объектов. Однако, как обратили внимание авторы работы [524], при использовании они подобны ужасным диаграммам потоков данных, хотя к ним и добавлены некоторые новые характеристики. Диаграммы видов деятельности уходят корнями к схемам событий Ptech и Martin/Odell. Состояния обычно рассматриваются как "состояния процесса". На рис. 8.11 показана диаграмма видов деятельности в области выполнения заказа. Из нее совсем не очевидно, о каком объекте идет речь, разве только что выполнение заказа само является объектом (целью) некоторого вида. Кое-где можно улучшить ситуацию путем добавления "плавательных дорожек", которые тоже показаны на этом рисунке.

На рисунке состояния процесса — это прямоугольники с округленными углами. Стрелками задаются (возможно, условные) потоки управления. Ромбики — это блоки ветвления, как на обычных блок-схемах. Утолщенные горизонтальные линии представляют точки разветвления и соединения потоков управления. Это гораздо лучше описывать с помощью пред- и постусловий, как это сделано для ветви подготовки заказа. Серым цветом показаны "плавательные дорожки" и их имена.

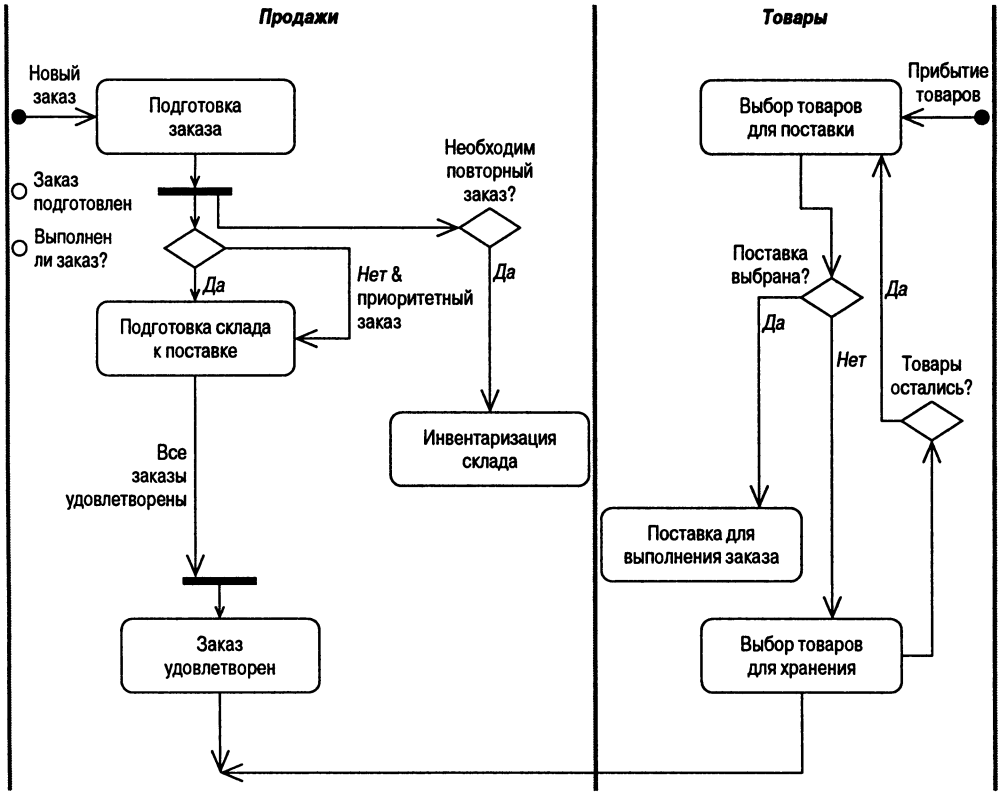


Рис. 8.11. Диаграмма видов деятельности по выполнению заказа

Проблема заключается в том, что без "плавательных дорожек" процессы полностью разрознены и не являются объектно-ориентированными вообще. Под "разрозненностью" понимается неинкапсулированность процесса в рамках некоторого объекта. Преимущество этого заключается в том, что не делается никаких предположений о распределении обязанностей. Однако это только видимость; диаграмма — это схема состояний, а состояния должны быть состояниями некоторого объекта (а здесь мы не можем сказать, какого). Использование "плавательных дорожек" делает наши предположения более строгими. Но даже здесь существуют проблемы. В известной работе [675] утверждается, что "плавательным дорожкам" не присуща семантика, они могут означать что угодно. Большинству людей хотелось бы использовать их для представления функциональных единиц в рамках организации.

Наши агентные диаграммы диалога являются удобной альтернативой диаграммам видов деятельности. Они легче для понимания пользователем, в том время как обозначения диаграммы видов деятельности трудны для запоминания, понимания и объяснения. Конечно, иногда диаграммы видов деятельности полезны, но таких случаев я нашел немного.

В [524] даются следующие критерии целесообразности использования этого вида представления состояний. Диаграммы видов деятельности можно использовать в следующих случаях.

- Состояние объекта является сложным и значительным (использование диаграмм состояний).
- Существуют сложные взаимодействия между несколькими объектами, которые могут вызывать изменения состояний друг друга. Это часто случается в системах управления сложными объектами.
- Поведение объекта реализуется под управлением событий в рамках одного потока. Объект имеет одну переменную состояния (общеизвестно, что бизнес-процессы являются многопоточными).
- Культура пользователей допускает их применение (как в области телекоммуникаций).

Необходимо избегать их использования при следующих условиях.

- Имеется несколько потоков (как в обычном бизнес-процессе).
- Имеются сложные взаимодействия между длинной последовательностью объектов.
- Объекты содержат несколько переменных состояния.

Я согласен с этими аргументами и считаю, что "согласованный" метод моделирования на основе агентов и прецедентов легок для понимания большинства людей, не связанных с техникой. Этот метод очень производителен в контексте перепроектирования или реорганизации бизнес-процессов. Еще одна причина непригодности диаграмм видов деятельности заключается в том, что очень трудно понять причины переходов между состояниями или способ их реализации.

Диаграммы видов деятельности напоминают диаграммы потоков данных, которые применялись в 1960-х годах прошлого столетия. Как мы обсудим ниже в этой главе, они полезны лишь для документирования или моделирования систем. Для этой же цели могут быть использованы диаграммы последовательностей, но диаграммы видов деятельности предпочтительнее, если мы хотим показать потоки управления или ветвление решений. Это приводит к довольно прагматическому решению относительно того, какие обозначения лучше подходят в каждом отдельном случае.

8.6. От диалогов к задачам и прецедентам

Моделирование задачи — это общий вариант моделирования прецедентов. Оно дает возможность реализовать агентную модель и модель эффективности пользовательской задачи очень кратким объектно-ориентированным способом. Теперь наше внимание будет сосредоточено, в основном, на этапе решения задачи в модели диалога. Мы покажем, что задачи можно представить как истинные объекты. Поэтому нам не нужны никакие новые обозначения; мы можем полностью рассчитывать на методы, описанные в главе 6. Такой подход противоположен методу, предполагающему применение UML. Язык UML представляет ряд новых обозначений для указания зависимостей в рамках прецедента, в том числе стереотипы `<<include>>` и `<<extend>>`. У нас имеется прекрасная возможность придерживаться известных представлений для объектов и основных четырех видов структурных связей UML:

классификация, композиция, использование и ассоциация[4]. Предлагаемая модель отличается субъектом, но структура понятий моделирования совпадает. Не надо забывать, что для моделирования действий задачи мы используем объектную метафору, поэтому прецеденты тоже рассматриваются как объекты. Отношения классификации и композиции позволяют группировать подобные объекты в рамках общей сети. Это относится и к задачам.

Декомпозиция задачи — это еще одна важная особенность процесса. Она позволяет выявить задачи, которые участвуют в решении нескольких задач более высокого уровня. Это обеспечивает повторное использование вложенных задач. Кроме того, подсчет "атомарных" задач внутри модели или *точек включения задач* является важным показателем возможностей системы и ее сложности. Эта и другие метрики обсуждаются в главе 9.

Операции объектов задачи — это методы, которые включают обработку исключительных ситуаций. Исключения, или "побочные сценарии", представляют собой необычные задачи, которые выполняются в определенных условиях. Часто оказывается, что обработка исключений допускает многократное использование в контексте ряда задач высокого уровня. Исключения могут быть вложенными: могут существовать исключительные ситуации для исключений. Например, мы обычно прерываем нормальную работу, беря летом отпуск. Но если возникают непредвиденные обстоятельства, такие как болезнь одного из членов семьи, мы приостанавливаем отпуск и мчимся домой.

Хотя мы моделируем задачи как объекты, их также можно рассматривать как аспекты поведения агента. Эта концепция аналогична понятию операции типа.

Теория сценариев

Задачи могут быть описаны несколькими способами, но наиболее полезно описывать задачу через ее сценарий. **Сценарий задачи** (task script) представляет собой *стереотипную* задачу. Это текстовое описание задачи (в идеале, простыми предложениями) со списком суперзадач, модульных задач и задач, которые описывают исключительные случаи (побочные сценарии). Это обеспечивает общую запись обработки исключительных ситуаций, которая кажется не естественной для пользователей метода прецедентов, где исключительные ситуации (расширения) "привязаны" к определенным прецедентам. В [417] прецедент определяется как "поведенчески связанная последовательность транзакций *в диалоге с системой*", а в [418] — как "последовательность транзакций *в системе*, задача которой состоит в достижении некоторого измеримого результата". Мое курсивное выделение указывает на незначительное противоречие между двумя определениями. Я буду исходить из общепринятого большинством пользователей соглашения, что задача описывает диалог в жизни, а не в системе и что прецеденты описывают только интерфейс системы. Сценарий задачи — это тоже часть модели мира, но отличие в том, что он не привязан к интерфейсу. Более того, важно, что сценарий задачи описывает *стереотипное* поведение в мире.

Например, рассмотрим сценарий задачи, описывающий посещение ресторана. Идея состоит в том, что каждый посетитель ресторана *всегда* выполняет одну и ту же последовательность действий.

1. Вход в ресторан
2. Вызов официанта

---

[4] Ден Ровсорн (Dan Rawsthorne) из компании Dimensions Inc. в частной беседе рассказал о собственной теории прецедентов (с целями), для представления которой он использует обозначения стандарта UML. Его семантика совпадает с моей. В [174] также приводятся доводы в пользу добавления целей к прецедентам.

3. Выбор места
4. Чтение меню
5. Выбор блюд
6. Принятие пищи
7. Оплата счета
8. Уход

Это несомненно хороший стереотип посещения ресторана в любой стране. Однако никакое посещение ресторана точно не следует этому сценарию. Например, возможен следующий ход событий.

1. Войти в ресторан
2. Позвать официанта
3. Идти к выбранному месту
4. Поскользнуться на шкурке банана ...

Сценарий нарушен и должен быть восстановлен, прежде чем клиент сможет утолить голод. Это выполняется с помощью вызова побочного сценария, который связан со стереотипными исключениями. **Побочный сценарий** (side-script) — это сценарий хорошо известной ситуации, выполняемый при генерации исключения в другом сценарии. Мысленно представим себе модель "передачи сообщения" побочному сценарию с просьбой о помощи. Далее выполняется побочный сценарий.

1. Встать
2. Отругать себя
3. Посмотреть вокруг, чтобы увидеть, кто смеется
4. Оскорбить их словесно или побить
5. Возвратиться к прерванной задаче

Важно отметить, что сценарий со "шкуркой банана" срабатывает не только в контексте посещения ресторана. Он будет работать везде: на улице, на фабрике или в офисе (если там нет ковров). Такие сценарии, как правило, сложны и могут быть пригодны для многих исключительных ситуаций. Например, что делать, если необходимо оплатить счет, а вы обнаруживаете, что потеряли бумажник. Сценарий описывает хорошо известную и стереотипную ситуацию, с которой приходится сталкиваться много раз. Для обработки таких исключительных ситуаций объекты задач могут посылать друг другу сообщения. Из этого вытекает, что можно классифицировать сценарии задач и выполнять их декомпозицию. Например, сценарий оплаты счетов за электричество, газ и воду может быть обобщен в сценарий задачи "оплата счета". Другими словами, его можно рассматривать как объект в рамках объектной модели. Эта объектная модель задачи не является моделью бизнес-объектов BOM (Business Object Model). Объектная модель задачи TOM (Task Object Model) — это модель мира, а модель бизнес-объектов — это модель создаваемой системы.

Идея сценариев задач во многом основана на теории сценариев искусственного интеллекта [683] и задаче анализа человеко-машинного интерфейса. Сценарии задач могут рассматриваться как общие прецеденты. Прецедент может состоять из одного предложения. Принимая во внимание атомарность задачи, сценарий должен состоять из единственного предложения. Попутно следует заметить, что побочный сценарий имеет точно такую же семантику, которая в [683] названа "субсценарием".

Итак, задачи могут быть описаны как *подлинные* объекты со своими атрибутами и связанными с их выполнением сообщениями и действиями. Наиболее общий метод анализа задачи, вытекающий из диалога, — это последовательная декомпозиция задачи вплоть до получения атомарных задач. При желании сценарии могут быть написаны с использованием грамматических конструкций вида: подлежащее-глагол-прямое дополнение-предлог-косвенное дополнение.

Декомпозицию задачи можно продолжать до *бесконечности,* добавляя большее количество деталей или перемещаясь к более низкому физическому уровню. Процесс останавливается, когда слова, которых нет в словаре обычного пользователя, будут представлены на следующей ступени разложения. Например, инструкция "клерк передвигает мышь к полю количества и вводит ..." не является атомарной, потому что слово "мышь" не является частью онтологии заказа. Мы должны остановиться на стадии "клерк вводит количество...". Другими словами, **атомарный** сценарий достигается в процессе декомпозиции объектов задач, когда выполнены следующие условия.

1. Сценарий задачи описывается простым предложением (в идеале, в формате SVDPI).
2. Дальнейшее разложение приводит к выражениям, которые не относятся к онтологии данной предметной области.

Задачи и прецеденты

Прецеденты были введены в качестве метода сбора объектно-ориентированных требований Якобсоном [417] и повсеместно признаны как блестящая и полезная идея. Большинство методов проектирования построено на основе прецедентов, которые представляют основной метод объектно-ориентированной разработки.

Сценарии задач впервые были введены в рамках метода SOMA и имеют три главных интеллектуальных прототипа: прецеденты Якобсона, иерархический анализ задач и сценарии из теории концептуальных зависимостей [683]. Понятие сценария используется в [672], хотя авторы этой работы пришли к нему иным путем.

На первый взгляд, метод прецедентов имеет более функциональный оттенок. Ведущие разработчики метода OMT даже включили обработку прецедентов в функциональную модель OMT, базирующуюся на потоках данных [82]. Это привело критиков к утверждению, что прецеденты вовсе не объектно-ориентированны. Однако когда появились прецеденты, то стало ясно, что объектно-ориентированного анализа и методов проектирования недостаточно для инженерии технических требований. Кроме того, большинство этих методов полностью основано на данных, что ведет к трудностям в общении с пользователями, для которых более удобны функционально-ориентированные описания. Пока прецеденты и метод Objectory, в рамках которого они использовались, были до некоторой степени "функциональны", это был небольшой шаг вперед. Проблема заключалась в объединении этого подхода с функциональной точкой зрения в рамках точного объектно-ориентированного, основанного на распределении обязанностей метода, а не в простом соединении с методом, основанным на данных.

Если рассмотреть проблему более серьезно, то отсутствие точного и универсального общепринятого определения приводит к распространению так называемых методов проектирования на основе "прецедентов". При этом каждая компания имеет собственную версию этой теории. На конференции по объектно-ориентированным технологиям, проведенной в Оксфорде (Англия) в 1996 году, участников спрашивали, использовали ли они прецеденты. Четырнадцати участникам, которые ответили утвердительно, предложили дать определение

прецедента. В результате было получено четырнадцать различных определений, хотя трое из респондентов работали в одной и той же компании (в разных отделах). В [174] описаны подобные результаты, полученные в США. Участники семинара по прецедентам в рамках конференции OOPSLA'97 (в том числе автор данной книги) пришли к необходимости формулировки точного определения прецедента, возможно даже под новым, более ясным названием (например, сценарий задачи). Однако уже существует торговая марка прецедента, и поэтому мы должны жить с этим ярлыком, несмотря на плохое определение. Я не согласен с таким пессимизмом и продолжаю предлагать термин *сценарий задачи*, подчеркивая свою глубокую признательность Якобсону (Jacobson). Точность имеет большое значение в науке и проектировании.

Мой опыт работы с прецедентами относится приблизительно к 1991 году, когда я попытался использовать их при проектировании требований для банковской системы заключения сделок. Я быстро пришел к выводу, что предметная область описывалась сотней прецедентов — слишком много для управления. При этом сценариев задач было одиннадцать — насколько лучше! Этот недостаток очень серьезен, хотя в некоторых областях (например, при проектировании оборудования торговых автоматов или переключателей) это не прослеживается. Во многих областях существуют сотни или даже тысячи прецедентов, потому что в бизнес-процессах имеется много исключительных ситуаций. Исключения — не "ошибки" компьютерных программ. Это важные, а часто критические варианты прецедентов. Они не являются конкретными сценариями, потому что сами выполняются во множестве конкретных сценариев, соответствующих реальному использованию. С точки зрения практики не существует никаких сомнений, какую методику нужно использовать в будущем, но возникает теоретический вопрос, почему число прецедентов растет экспоненциально. На это существует несколько причин[5].

В частности, мы хотим трактовать прецеденты как обычные объекты, чтобы иметь возможность повторно использовать их, как мы это делали в главе 6. Это оказывается проблематичным из-за неоднородности типичных прецедентов. Из-за недостаточной проработки семантики прецедент может охватывать несколько взаимосвязанных, но независимых задач.

При попытке рассматривать прецеденты UML в качестве объектов возникают серьезные проблемы (это отмечает сам Якобсон). Одна структурная связь между прецедентами, а именно стереотип `<<extends>>`, не согласуется с принципом инкапсуляции (формирования пакетов). Поэтому мы не можем рассматривать прецеденты как *истинные* объекты. Из этого вытекает, что они не пригодны для повторного использования. Если мы согласимся, что прецеденты являются подлинными объектами, то у нас не будет необходимости добавлять стереотипы `<<extends>>` и `<<includes>>`; мы просто сможем использовать уже имеющиеся и точно определенные объектно-ориентированные связи, такие как композиция и использование, для охвата той же семантики и, кроме того, получим обозначения для наследования и ассоциации. Отношение использования упрощает управление исключительными ситуациями.

---

[5] Это нарушение принципа бритвы Оккама (Ockham) впервые было замечено на выставке Object Expo Europe в 1993 году. Оно подтверждается анекдотическими фактами из нескольких известных мне проектов и наблюдениями других участников.

Стереотипы <<includes>> и <<extends>>

Один прецедент "использует" или "включает" другой, если последний — "часть собственного описания" [418].

Таким образом, отношение использования в методе Objectory (аналогично стереотипу UML `<<includes>>`) несомненно соответствует декомпозиции задачи и отношению агрегации в UML. Различие состоит лишь в интерпретации. В Objectory не подчеркивается применение прецедентов для декомпозиции задачи.

Говорят, что прецедент "расширяет" другой, если он может быть "вставлен внутрь" последнего [417]. Таким образом, "расширения" связаны с побочными сценариями, хотя стрелки указывают в противоположном направлении для сохранения инкапсуляции. Расширение "не знает", что является частью чего-то большего.

Для иллюстрации используем пример из [417], где рассматривается проектное решение торгового автомата, в котором допускается повторное использование контейнеров. Прецедент `Returning item` (Возврат тары) описывает вставку клиентом использованного контейнера в машину. Прецедент `Item` (Контейнер) расширяет предыдущий и описывает последовательность событий в случае, если контейнер заклинит в желобе. Однако он выполняется только в особых случаях, в то время как сценарий из модели ресторана `BananaSkin` (Шкурка банана), описанный выше, — в большинстве исключительных ситуаций. Одно из следствий этого заключается в том, что прецедент `EnterRestaurant` (Поход в ресторан) может выполнять поиск соответствующего сценария типа `BananaSkin`, а не статически хранить ссылку на него. Таким образом, для инкапсуляции и повторного использования важно, чтобы побочный сценарий не знал, какие сценарии он "расширяет".

Читателя могут ввести в заблуждение описания побочных сценариев с учетом связей "включения" и "расширения" UML. Внимательное чтение работы Якобсона (Jacobson) и литературы по UML дает возможность сделать вывод о некоторой, хотя и неполной аналогии между этими понятиями. Нашей задачей является описание общей концепции моделирования объектов. Тогда разработчику придется выучить только один набор терминов и понятий, которые он сможет применять для моделирования как задач, так и бизнес-объектов.

Атомарность

Как руководители проекта, мы хотели бы надеяться, что ряд прецедентов при анализе требований принесет некоторую пользу для окончательной системы или хотя бы для попыток ее построения. Это поможет определить стоимость продукта и самого проекта. Общеизвестно, однако, что прецеденты сложно использовать в качестве метрики, так как они могут иметь любую продолжительность.

Для моделей прецедентов невозможно разумно определить никакую метрику, не вводя понятия *атомарности,* как в действительности сделали многие компании. Понятие сценария задачи включает понятие атомарности, с помощью которого разработчик определяет сложность задачи простым подсчетом атомарных сценариев, состоящих из простых предложений.

Объединение наборов задач

Прецедент охватывает последовательность нескольких, возможно, равнозначных задач. Например, рассмотрим систему автоматизации торговли, которая автоматически вычисляет стоимость заказов или отклоняет их, исходя из некоторых критериев. На самом высоком уровне абстракции было бы уместным иметь четыре прецедента: автоматическое назначение

цены, автоматическая оценка стоимости заказа, оценка стоимости заказа вручную и задание цены вручную. Это не экономно, поскольку в разных прецедентах могут встречаться одни и те же фрагменты. Лучше иметь независимые сценарии задач для различных компонентов прецедента. Потом их можно собрать в полной модели бизнес-процесса. Это способствует повторному использованию объектов задач.

Ограничение интерфейса

Прецеденты подвергаются критике за учет только внешних аспектов интерфейса системы, хотя они успешно использовались во многих проектах. Это, конечно, относится ко всем интерфейсам, а не только к человеко-машинному.

Этот так называемый недостаток является преднамеренным. Замысел авторов прецедентов заключался в концентрации внимания разработчиков на том, *что* делает система для своих пользователей, а не на том, *как* она это делает. В некоторых случаях это оправдано и допустимо при создании телекоммуникационных переключателей и контроллеров процессов, но во многих системах, особенно заменяющих бумажные потоки, вопрос о принципах работы системы возникает на самых ранних стадиях проектирования. Он часто является основой понимания бизнес-процессов и представляет благоприятную возможность для их реорганизации. При таких обстоятельствах предпочтительнее использовать вариант прецедента, который допускает внутреннее моделирование.

Это замечание противоположно предыдущему, которое сводилось к тому, что анализ прецедентов *не выходит за рамки* системы. Здесь же проблема заключается в том, что он не смотрит *вовнутрь*. Мы, конечно, не предлагаем нарушать инкапсуляцию.

Объекты-контроллеры

Наши критические замечания в адрес OOSE не ограничиваются проблемами с прецедентами, методом Objectory и языком UML. В методе OOSE рекомендуется связывать прецеденты с тремя видами объектов реализации: объектами сущностей, интерфейсными объектами и объектами-контроллерами. Введение объекта-контроллера в процесс анализа может серьезно нарушить принцип инкапсуляции и привести в дальнейшем к сложным проблемам эксплуатации системы. Это происходит потому, что контроллеры часто выполняют роль главной программы (особенно в системах управления на основе данных) и должны иметь много информации о других объектах, которыми они управляют. Таким образом, в случае изменения одного из этих объектов контроллер должен быть модифицирован. Стиль реализации, при котором объект-контроллер имеет доступ к нескольким объектам сущностей (т.е. большому количеству данных), был подвергнут критике в работе [699], обсуждавшейся в разделе 6.2.1. Как указывается в главах 6 и 10, гораздо лучше использовать агенты, основанные на правилах.

Прецеденты и сценарии

Среди пользователей методов Objectory и OOSE, а также языка UML имеются некоторые противоречия в трактовке термина "сценарий". В связи с этим возникают проблемы и с применением термина "прецедент". В 1994 году на конференции OOPSLA группой специалистов был поднят вопрос о взаимосвязях между сценариями и прецедентами. Якобсон (Jacobson) и Буч (Booch) согласились, что сценарий является "отдельным экземпляром прецедента", но этот вопрос в дальнейшем детально не прорабатывался. Это различие гораздо легче понять, если руководствоваться отличиями между абстрактными и конкретными понятиями, а не

между классами и экземплярами. И прецеденты, и сценарии относятся к уровню экземпляров. Единственное различие заключается в том, что сценарий включает порядок следования информации.

Метод прецедентов предлагает разработчикам начать с обнаружения поведенческих родственных последовательностей транзакций в диалоге с системой. Эти описания служат основой для создания иерархии классов и реализации методов в будущей системе. Несколько других подходов связано с изучением сценариев использования системы. В большинстве случаев сценарий более детализирован и конкретизирован, чем прецеденты, и может быть связан с конкретным исполнителем. Таким образом, выражение "пользователь вводит число" является прецедентом, в то время как выражение "Джон вводит возраст своей жены" есть сценарием; это минимум, который достаточен для понимания целей этой книги.

Базовые или общие прецеденты

Еще одной причиной выявления слишком большого числа прецедентов является отсутствие определений существенности и общности (универсальности).

Понятия сценариев задачи и базовых прецедентов, введенных Константином (Constantine), во многом перекрываются.

Базовый прецедент отличается от детального, а также от сценария задачи. В качестве примера рассмотрим прецедент, связанный с получением денег из банкомата ATM (Automatic Teller Machine). Сначала в автомат вставляется карточка, вводится личный номер и т.д. Базовым прецедентом в этом примере будет получение денег. Оценим атомарность задачи. Уровень атомарности зависит от цели описания и предполагает отсутствие терминов, не относящихся к предметной области. В [183] базовый прецедент определяется так.

> "Это абстрактный прецедент, который описывает общую структуру прецедентов, представляет намерения или цели пользователей, выполняющих некоторые роли, явно и обобщенно представляет основное ядро того, что пользователи хотят или что им необходимо выполнить, независимо от реализации в определенном интерфейсе пользователя. Базовый прецедент выражается в терминах предметной области пользователя и предполагает идеализированную технологию или не зависит от технологии".

Другими словами, базовый прецедент — это структурное повествование, выраженное на языке, который могут понять пользователи. Он представляет собой простое, абстрактное, независимое от технологии описание задач пользователя, выражающее цель, лежащую в основе взаимодействия. Эти определения объединяют то, что я назвал "общностью" и "атомарностью". Я думаю, будет лучше разделить эти концепции. Как я покажу в следующем разделе, сценарий задачи представляет собой общий (родовой) прецедент. Соответственно, сценарий задачи является атомарным и должен быть назван базовым. Оказывается, что базовый сценарий задачи соответствует понятию базового прецедента. Выявление базового прецедента — это ключевая часть метода, описанного в [184].

Теперь необходимо определить различия и связи между понятиями действий, прецедентов и сценариев. Для этого нужно обобщить[6] идею прецедента. Действия позволяют разработчикам моделировать внутреннюю организацию бизнес-процесса.

Как отмечалось ранее, сценарий видится как экземпляр прецедента: он описывает его фактическую реализацию. Прецедент может быть задуман как пример связанного набора

[6] Иными словами, выработать общее представление.

действий: он описывает типичный ход событий бизнес-процесса. К сожалению, эта терминология не совершенна, потому что сценарий представляет собой экземпляр класса сценариев, а класс всех сценариев не будет соответствовать прецеденту. Идея, однако, правильная и может быть перефразирована следующим образом: действия представляют общий прецедент или набор сценариев задачи (*эквивалент* класса прецедентов). Таким образом, прецедент — это общий сценарий.

Теперь можно рассматривать **прецедент** как эквивалент класса сценариев. Действие — это **общий** или **базовый** сегмент прецедента. Подобным образом прецедент представляет собой экземпляр общего сценария. Прецедент — это эквивалент набора сценариев, а последовательность действий задачи — эквивалент набора прецедентов. Отношения эквивалентности легко определяются неформально, но часто могут быть неточными [824]. Например, можно считать эквивалентными все сценарии, в которых некоторый индивидуум вводит номер телефона другого лица, или все прецеденты "принятие еды вне дома". Неточность этих описаний можно проиллюстрировать сценарием посещения ресторана и прецедентом посещения учреждения быстрого питания, имеющего слабое отношение к любой нормальной концепции ресторана. Фактически, употребляя термин ресторан, мы допускаем *нечеткость*, потому что возникает диссонанс, когда мы пытаемся применить слово РЕСТОРАН к заведению McDonald's. Сценарий в этом случае становится не пригодным.

Три уровня абстракции соответствуют (неформально) трем традиционным уровням моделирования информационных систем: уровням данных, информации и знаний. Однако, по мнению Якобсона[7] (Jacobson), по этим уровням нельзя различать абстрактные и конкретные прецеденты. Это разграничение позволяет только классифицировать или обобщить сценарии и прецеденты.

Как мы видели ранее, обобщение в большинстве случаев означает абстрагирование от определенных исключений, число которых зачастую слишком велико.

Что может быть сделано для урегулирования всех этих проблем? Необходимо обоснование метода, подведение теоретической базы и точное определение. Нам необходимы понятия *общности* и *атомарности* для прецедентов или их эквивалентов. Прецеденты не имеют ясной связи с моделью бизнес-процессов и предлагают в качестве такой модели свою собственную, которая не очень подходит по смыслу. Нам необходим метод, обеспечивающий объектно-ориентированные средства поддержки инкапсуляции и наследования. Метод моделирования SOMA удовлетворяет всем этим требованиям.

Размышления о проблемах инженерии требований отражают различия между методами Objectory и SOMA, а также различия между обычными пользователями, работающими в разных областях. Для инженеров телекоммуникационных систем обычно достаточно детальной спецификации. Им удобно работать с описаниями состояний автомата. Банкирам этого недостаточно. Дальнейшее практическое следствие заключается в значительном уменьшении размеров модели. Гради Буч (Booch), накопивший опыт разработки систем, заметил, что: "Большинство систем характеризуется 10 прецедентами [высокого уровня], состоящими из последовательности более важных прецедентов и [другой] последовательности вторичных прецедентов" [345].

Главная причина такого бурного роста заключается в отсутствии общих методов обработки исключений и в природе понятий "включения" и "расширения", рассмотренных выше.

---

[7] В частной беседе.

8.7. От объектной модели задачи к объектной модели бизнес-процессов

В этом разделе я объясняю, как на основе сценария задачи может быть получена спецификация UML или объектная модель бизнес-процессов.

Вспомним, что внешние агенты отличаются от внутренних по двум причинам. Внутренние агенты работают в рамках бизнес-процессов, и мы достаточно знаем о задачах, которые они выполняют. Внешние агенты могут выполнять задачи, о которых мы ничего не знаем и которые могут нанести нам ущерб, потому что ведут к передаче сообщений или **инициализации событий**. В случае инициирования сообщений исполнителями или внутренними агентами мы обычно знаем причины событий, потому что они представляют собой результат выполнения других задач. Когда инициатором выступает внешний агент, мы почти всегда испытываем недостаток знаний, и инициированное событие воспринимается как данность. Таким образом, **исполнители** и пользователи системы (выполняющие определенные роли) могут быть внутренними или внешними агентами, но обычно внутренними. Следовательно, мы можем легко спутать внутренних агентов с исполнителями и использовать не то обозначение UML. Сообщения всегда приводят к инициализации событий, хотя для внутренних агентов мы обычно знаем задачу, которая привела к реализации данного события.

Упрощенные агенты

При объектно-ориентированной разработке агенты (пользователи и клерки) обычно представляются в объектной модели бизнес-процессов довольно неинтеллектуальными объектами, которые выполняют лишь несколько операций (или совсем бездействуют), но содержат несколько типичных атрибутов. Важно не путать эти внутренние представления, которые отвечают лишь за хранение статических данных, с их реальными прототипами, которые обладают сложным поведением. Такие внутренние представления агентов являются внешними по отношению к системе и почти всегда устойчивы. Обычно в объектно-ориентированной модели интеллектуальные клиенты и клерки представляются простыми бессловесными структурами данных, а заказы, которые в реальности представляют собой клочки бумаги, — "умными" объектами, отвечающими за выполнение нескольких действий, таких как утверждение лимитов пользователей или сравнение котировок акций. Существует мощная альтернатива этому подходу — представлять в системе интеллектуальные сущности *интеллектуальными агентами* (intelligent agent).

На рис. 8.12 внутренний агент (исполнитель или клерк) получает заказ от потребителя и вводит его в систему автоматизации бизнес-процесса. Потребитель инициирует заказ под воздействием некоторых условий. Конечно, такая система, скорее всего, реализуется как компьютерная, но с таким же успехом это может быть картотека или нечто подобное. Наша задача, как аналитиков, — определить объекты бизнес-процесса. К ним несомненно относятся клерк, потребитель, изделие и, конечно, заказ. Является ли кредитный лимит атрибутом клиента или он относится к агенту — менеджеру по кредитам? В банковских системах обычно реализуется второе предположение. Теперь с агентно-ориентированных позиций необходимо ответить на вопрос: насколько интеллектуальным является сам объект заказа? Относятся ли правила оценки кредитов к классу `Orders` (Заказы)?

Типичный разработчик объектно-ориентированных систем наверняка захочет делегировать правила и большинство обязанностей классу `Orders`. Но имеет ли это смысл для обычного

пользователя? Я не думаю. В реальном мире клиенты и клерки обладают определенным интеллектом и поведением. Заказ представляет собой безжизненный, бесплатный клочок бумаги. Но в наших системах основная роль отводится заказам, а не клиентам и клеркам, которые обычно представляются в таблицах баз данных, где хранится информация о них (в виде имени, адреса и кода для входа в систему). Функции большинства таких объектов сводятся к определению текущего состояния счета на основе данных других классов, таких как `Orders` (Заказы). С одной стороны, это целесообразно: реальные клиенты могут размещать заказы, но не стоит передавать им управление системой. Бизнес-логика при этом очень проста, если исключить моменты возможной неуплаты или обвинения в мошенничестве.

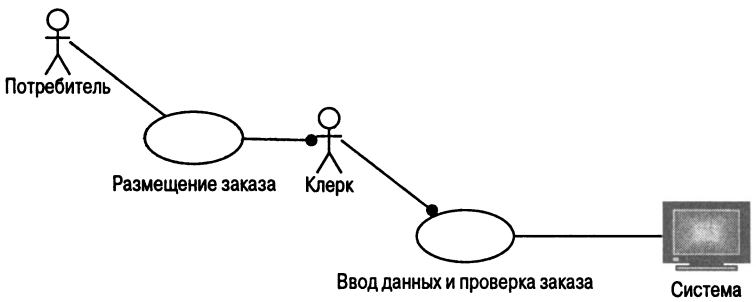


Рис. 8.12. Процесс ввода заказа

Можно пойти другим путем: ввести агент `Clerk's Assistant` (Ассистент клерка) и возложить на него проверку правильности заказа и его подтверждение, передав ему соответствующие обязанности объекта `Order` (Заказ). Здесь имеется три значительных преимущества.

1. В распределенной системе агент помощника может быть мобильным. Это позволит уменьшить сетевой трафик и улучшить общую структуру управления.
2. Разгрузка объекта `Order` позволяет уменьшить несоответствие между нашей новой системой и прежними базами данных, в которых объект `Order` все еще сохранен.
3. Уменьшается "когнитивное несоответствие" между системой и миром. Это ведет к облегчению диалога между пользователями и разработчиками, так как они используют общую модель приложения. Интеллектуальные клерки в офисе моделируются, как интеллектуальные сущности в компьютере (или почти так, потому что реально разумных компьютеров не существует и не может существовать). Электронные заказы столь же неактивны, как и их бумажные аналоги.

Применимость этого метода полностью зависит от приложения и его среды. Однако знание того, что это возможно, дает разработчикам дополнительный инструмент. В следующий раз они будут моделировать систему с учетом и этой альтернативы.

Хотя добавление наборов правил в объекты SOMA никак не связывалось с агентной технологией, это дополнение казалось довольно хорошим средством создания интеллектуальных моделей, в том числе на основе мобильных агентов. Такая мотивация впервые появилась в 1989 году в процессе работы над проектом по реорганизации бизнес-процессов. Теперь ясно, почему для моделирования системных агентов необходимо точно такое же расширение

объектно-ориентированного анализа, как и для моделирования бизнес-процессов. Эти объекты моделирования схожи между собой. К теме агентно-ориентированных систем мы возвратимся в главе 10.

Для моделирования обязанностей клерка в торговой организации можно использовать инкапсулированные (или **упрощенные**) агенты. Стереотип агента можно рассматривать как упрощенную модель реальных агентов в деловом мире. Если в системе используются агенты, представляющие бизнес-логику, то заказ моделируется как структура данных, более точно соответствующая представлению в существующей базе данных заказов.

Однако мы будем использовать более традиционный подход к моделированию, предполагающий интеллектуальность класса заказов. Рассмотрим сообщение `Enter&Validate Order` (Ввести и проверить заказ) более подробно. Детальное описание этого сообщения может основываться на структуре, размер которой существенно превышает представленные ранее в этой главе диалоги. Окно диалога может выглядеть так, как показано на рис. 8.13.

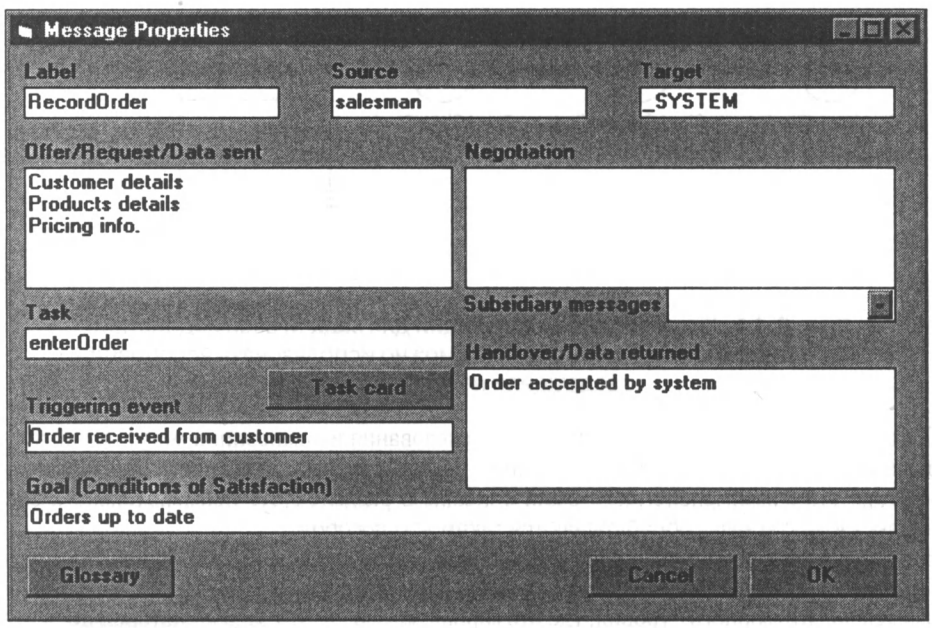


Рис. 8.13. Пример окна диалога

Процесс декомпозиции можно представить в графической форме (рис. 8.14). Каждая задача имеет свою сложность и требует некоторого времени. Обычно описывают только атомарные задачи, так как с их помощью можно представить любые сложные задачи. Затраченное время вычисляется как сумма времени, затраченного на выполнение компонентов на каждом следующем уровне. Сложность тоже является аддитивной характеристикой, если не указано иное.

Ассоциации задачи могут быть использованы для представления последовательности и координации описываемых задач. Их также можно эффективно применять для представления объединения задач и формирования полного бизнес-процесса. Обычно последовательность

задач читается слева направо, а распараллеливание отсутствует. Выразительной мощности ассоциаций задач и наборов правил достаточно для представления отношений любого уровня сложности. Более детально эти вопросы рассмотрены в разделе 8.11.

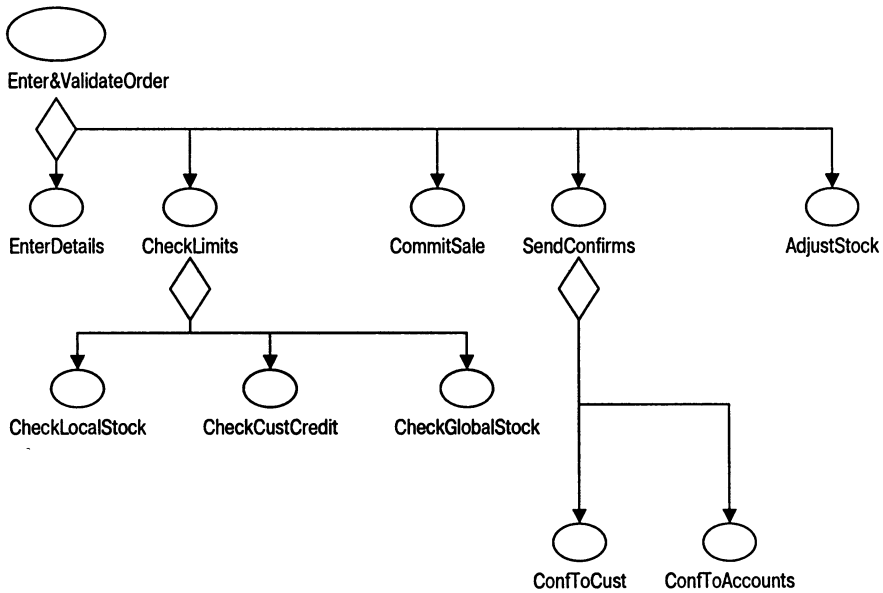


Рис. 8.14. Диаграмма декомпозиции для задачи взаимодействия. Для представления параллелизма можно использовать ассоциации задачи или наборы правил

При описании взаимодействия структур наследования и композиции могут возникнуть некоторые сложности. Однако такая ситуация редко встречается на практике. Правила для управления этим взаимодействием были описаны в разделе 6.3.1 применительно к бизнес-объектам. Объекты задач обрабатываются таким же способом.

В завершение анализа моделей процессов, бизнес-процессов и сообщений рассмотрим набор задач, полученных в результате декомпозиции. Предполагается, что декомпозиция продолжается до "атомарного" уровня, где атомарная задача — это задача, которая не подлежит дальнейшему разложению без введения терминов, не свойственных предметной области. Каждая атомарная задача представляется в форме стандартного предложения SVDPI (Subject/Verb/DirectObjects/Preposition/IndirectObjects — Подлежащее/Сказуемое/Прямые дополнения/Предлоги/Непрямые дополнения). Затем предложения в модели задачи анализируются (предпочтительнее это делать во время семинаров с пользователями) с целью выявления бизнес-объектов, на которых будет базироваться модель системы. Обычно анализ текста проводится по следующей методике: существительные соответствуют предполагаемым объектам, а глаголы — операциям. К сожалению, этот процесс является не автоматическим, а творческим. Обнаруженные таким образом классы могут быть описаны с помощью карточек классов (рис. 8.15) для использования в ролевых играх. В рамках этого микропроцесса создается модель бизнес-объектов BOM (Business Object Model), на основе которой и выполняется проектирование системы.

Имя класса *Product*	*~~Абстрактный~~/Конкретный* *Предметная область /Приложение/~~Интерфейс~~*
Описание/Ключевые слова *Товар характеризуется кодом, наименованием, описанием и ценой*	
Суперклассы/Интерфейсы: *Commodity*	
Классы компонентов:	
Атрибуты и ассоциации *ProductName* *ProductDescription* *ProductPrice* *ProductID* *StockLevel*	
Операции	Пары отправитель-сообщение
SetProductCode *Ввести код товара и другую информацию в базу данных системы*	*Product - CreateProduct*
ProductMargin	
Наборы правил/Инварианты *ProductPricingPolicy*	

*Ненужное зачеркнуть

Рис. 8.15. Карточка класса в методе SOMA

В результате анализа строятся диаграммы последовательностей или видов деятельности UML, которые эффективно описывают действия, выполняемые бизнес-объектами. В процессе проверки этих двух моделей на непротиворечивость (согласованность) и полноту иногда возникает необходимость "отладки" модели. Такой критический анализ можно выполнить с помощью диаграмм последовательности и видов деятельности либо вручную, либо с использованием обычных CASE-средств. Эти процедуры впоследствии составят основу сценариев тестирования системы.

8.8. Незаметность для пользователя

Важно понимать, что при переходе от модели бизнес-процесса к модели системы мы пересекаем Рубикон. Назад дороги нет! Если мы изменим нашу модель системы, скажем, в результате усовершенствования кода, то не сможем определить влияние такого изменения на модель бизнес-процесса. В SOMA каждое дерево задачи соответствует плану бизнес-процесса. Корневой узел дерева задачи служит контейнером для атомарных задач, которые составляют детали планов. Необходимо создать набор классов в модели системы (BOM), который поможет выполнить этот план. Такая реализация должна с чего-то начинаться. Поэтому каждая "корневая" задача соответствует **ровно одной** системной операции одного класса. Эта операция инициализирует системный процесс, который осуществляет план. Идентичность этого класса и операции записана как часть свойств задачи. Это дает возможность полностью отслеживать события и *удостовериться* в том, что система позволяет реализовать все указанные бизнес-процессы. Таким образом, наш ориентированный на задачу метод обеспечивает незаметность (seamlessness) для пользователя при переходе от бизнес-модели к модели системы, что невозможно в методах, основанных на прецедентах. Это также дает преимущества при тестировании, поскольку последовательности событий могут рассматриваться как проверочные сценарии.

Вниманию читателя были представлены две объектные модели метода SOMA: объектная модель задачи TOM (Task Object Model) и объектная модель бизнес-процессов BOM (Business Object Model). Отмечено, что они могут быть связаны с пользователями и разработчиками с помощью анализа карт CRC. Теперь я хочу показать, как эти взаимосвязи могут быть автоматизированы для обеспечения гладкой связи между системой и ее требованиями. Эта связь позволяет исследовать влияние возможных изменений системы на процессы и бизнес-цели. Кроме того, с помощью этой связи можно доказать, что спецификация соответствует требованиям.

В методе SOMA принцип объектного моделирования используется для построения нескольких моделей. Например, жизненный цикл процесса моделируется как цепочка объектов видов деятельности и соглашений между ними. Объектная модель агента — это модель бизнес-процесса. Объектная модель задачи — модель задачи, выполняемой пользователями (и другими агентами) как часть их бизнес-процессов. Объектная бизнес-модель — это модель объектов предметной области, которые составляют спецификацию компьютерной системы. В этом смысле объектную модель задачи можно считать частью *модели мира*. BOM может быть названа *объектной моделью системы*. Есть и более универсальная модель системы, которая строится позже и называется объектной моделью реализации. В качестве альтернативы можно создать выполняемую спецификацию с помощью средств генерации кода.

В то время как обычные методы предлагают различные средства моделирования для жизненного цикла почти каждого вида деятельности, объектно-ориентированные методы не разграничивают процессы анализа и логического проектирования. "Водораздел" может проходить между логическим и физическим проектными решениями, когда вводятся зависимые от языка детали, но использование такого языка, как Eiffel, устраняет это разграничение. Таким образом, объектно-ориентированная разработка должна быть "бесшовной". Однако при прямом переходе от модели требований к модели системы возникает настоящая пропасть, которую можно только перепрыгнуть. Используя простой пример, я попытаюсь объяснить, как решить эту проблему.

Пример

В подходе SOMA к инженерии объектно-ориентированных требований для бизнес-области сначала устанавливается миссия проекта, а затем вырабатывается ряд конкретных, измеримых целей с приоритетами. Затем строится модель бизнес-процесса или контекстная модель, отображающая внешних агентов (заинтересованных лиц из предметной области), внутренних агентов и агентов поддержки системы. После этого вводятся сообщения, отражающие диалоги между этими агентами. Из рис. 8.16 видно, как это можно применить к системе покупки иностранной валюты. На этом рисунке внешний агент или контрпартнер[8] передает сообщение внутреннему агенту (дилеру), приглашая его совершить валютную сделку. Дилер должен оговорить сроки, проверить условия сделки и войти в систему. Это представляется сообщением `ввод соглашения`.

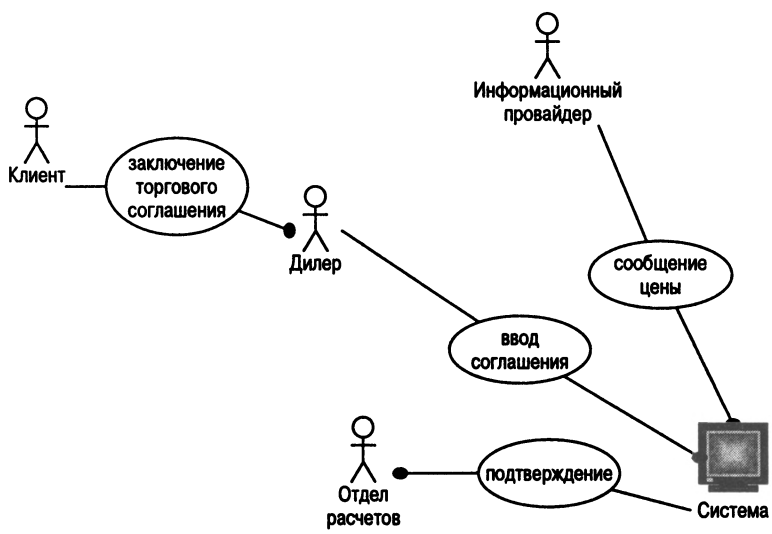


Рис. 8.16. Модель бизнес-процесса торговой сделки

[8] Клиента банка часто называют контпартнером.

У каждого сообщения должна быть своя цель и задача, которая называется корневой, потому что она находится в корне дерева композиции. Дерево композиции для сообщения `ввод соглашения` показано на рис. 8.17. Задачи для каждой цели анализируются и разлагаются до получения атомарных задач, которые являются листьями этого дерева. Затем описываются сценарии для атомарных задач.

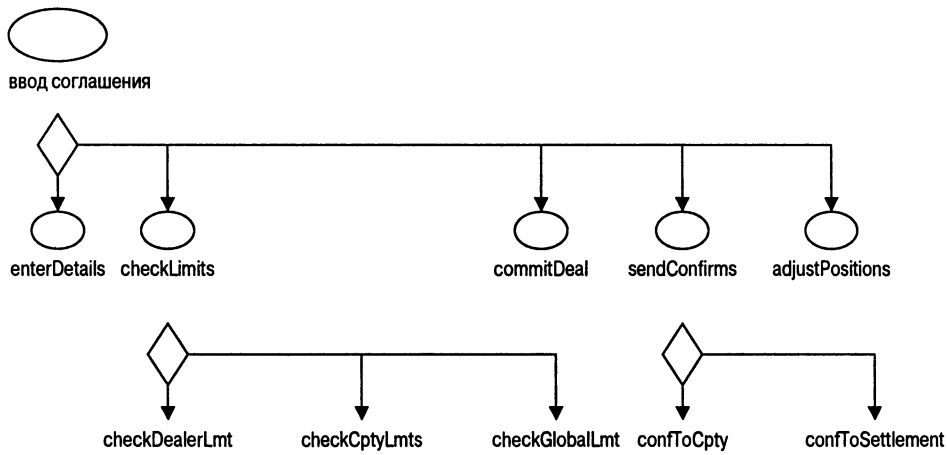


Рис. 8.17. Декомпозиция прецедента ввода соглашения

На рис. 8.18 показан атомарный сценарий для задачи `EnterDetails` (Ввод детальной информации). В этом сценарии упоминается ряд классов, например `Instruments` (Средства), `Counterparties` (Клиент) и (неявно) `Deal` (Сделка). Также имеется несколько явных атрибутов, таких как `Buy/Sell` (Покупка/Продажа). И наконец, упоминается операция `enter` (ввод). После предварительного анализа этого и всех других сценариев проекта системы становится ясно, что задача `enter deal` (ввод соглашения) соответствует обязанности `captureDeal` (заключение сделки) класса бизнес-объекта `Deal` (Сделка).

Представим эту обязанность как операцию `ImplementedBy` (Реализовать посредством) корневого объекта задачи `enter deal`, обеспечив постоянную связь между плановым процессом, представленным корневой задачей, и программным обеспечением, которое поможет пользователю выполнить этот план. Эта связь позволяет реализовать спецификацию.

Дилер вводит следующие данные: имя клиента, средство, количество, цена, покупка или продажа, специальные условия сделки

Рис. 8.18. Сценарий атомарной задачи `EnterDetails`

Связь задач с классами

Еще один взгляд на процесс спецификации схематично представлен на рис. 8.19, где показана миссия проекта, а также выделены соответствующие подцели. Каждая цель связана с несколькими коммуникационными действиями (диалогами) в модели. Каждый диалог имеет

определенную цель и определенную корневую задачу. Корневой задаче соответствует одна или более атомарных задач. Все эти связи полностью прослеживаются. Поэтому объектную модель задач можно строить итерационно.

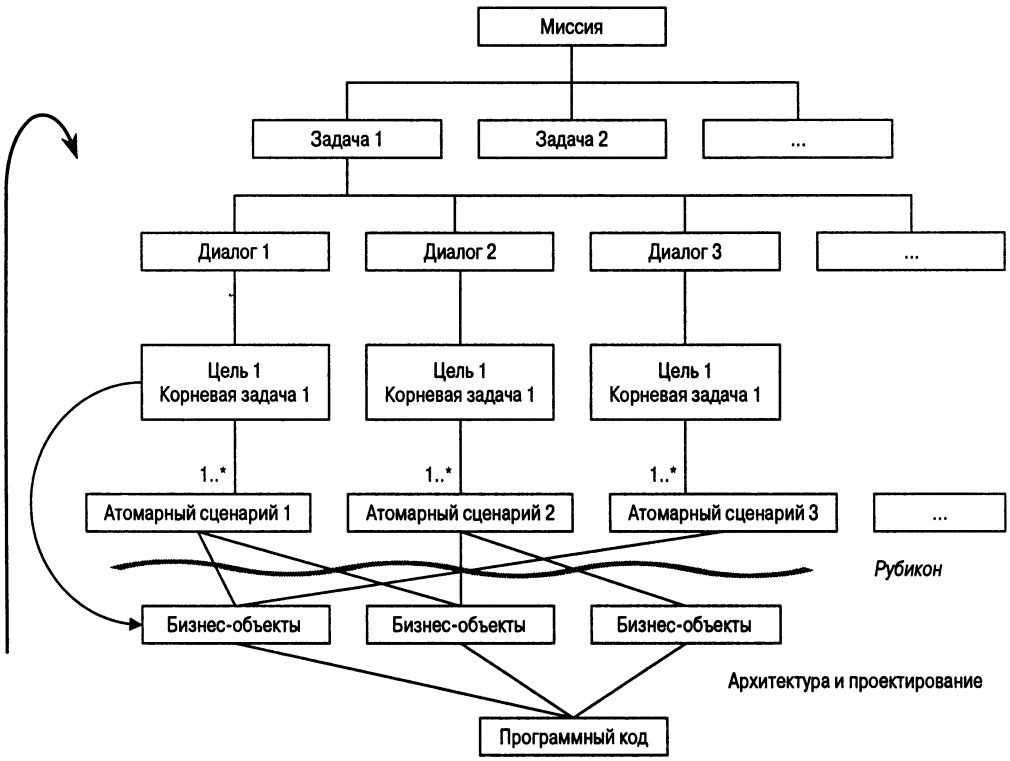


Рис. 8.19. Гладкий процесс

Теперь перейдем от модели мира к модели системы. Мы определили бизнес-объекты на основе существительных, найденных в сценариях. Затем были определены структуры классификации, композиции и ассоциации. Для каждого класса были выделены обязанности и правила с учетом глаголов, задействованных в сценариях. Это творческий процесс, который не поддается автоматизации. При этом мы перешли Рубикон и утратили возможность обратной трассировки.

Однако мы можем проверить взаимную согласованность двух моделей с помощью группы пользователей и разработчиков. Карточки классов (отражающие роли классов) позволяют показать, что все задачи реализованы правильно.

Как нам удалось незаметно перейти от модели мира к модели системы? Дело в том, что деревья задач содержат "планы" взаимодействия объектов во время выполнения задачи и таким образом описывают структуру системы и ее функционирование. Каждая корневая задача соответствует *ровно одной* системной операции класса, который инициирует выполнение плана. Наличие этой связи означает, что мы можем автоматически отслеживать последовательности событий. Другими словами, мы естественным образом построили связь миссии с

кодом и *обратно*! Теперь можем усовершенствовать модель бизнес-объектов и сгенерировать рабочий код (выполняемую программу), а затем проследить изменения в обратную сторону: от кода к операциям в моделях задач и даже к миссии. Для объектного моделирования могут быть использованы диаграммы видов деятельности или диаграммы последовательностей, основанные на модели прецедентов.

Ранее был введен атрибут задачи `ImplementedBy` (Реализовать посредством), обеспечивающий реализацию класса описываемой задачи. Так как эти две предметные области не являются прямым отображением друг друга, взаимосвязь между ними достигается за счет связывания простых действий и определения операций класса реализации, обеспечивающих системную поддержку задачи. При первом документировании задачи эти данные не известны. Мы можем вернуться к описанию задачи и заполнить эти данные только после построения модели бизнес-объектов. Такой замкнутый цикл очень важен в общем процессе. Он обеспечивает средства перехода от моделирования к реализации и проверку правильности модели.

Преимущества этого метода для разработки программного обеспечения в смысле качества и тестирования, я думаю, очевидны.

1. Последовательности событий позволяют строить сценарии тестирования системы.
2. Участие пользователей в создании этих последовательностей обеспечивает соответствие объектной модели бизнес-процессов и объектной модели задач.

Второе преимущество позволяет разработать методику для "доказательства" корректности спецификации системы и ее соответствия модели требований. Обычно понятие корректности применяется только в смысле соответствия реализации и спецификации. Наш метод позволяет объединить две технологии.

Строго говоря, описанная методика обеспечивает *проверку* объектной модели бизнес-процессов, а не доказательство ее корректности в математическом смысле. Это связано с нашим желанием установить взаимопонимание между пользователями и разработчиками под лозунгом: "Не доказывайте, что это работает, а убедите меня". Проверка на основе сценариев задачи имеет смысл только в том случае, если исследуется каждый сценарий. Поэтому попытаемся максимизировать полноту набора сценариев.

Схемы прецедентов, показанные на рис. 8.14 и 8.17, во многом схожи между собой. Зачастую такую организацию называют *шаблоном* (pattern), хотя данный шаблон больше относится к бизнес-процессам, а не к анализу или проектированию системы, с которыми связаны наиболее известные шаблоны.

8.9. Шаблон силлогизма для генерации прецедентов

На семинарах SOMA появился совершенно новый вид шаблонов — шаблоны извлечения знаний. На таких семинарах пользователи исследуют динамику системы с помощью карточек классов, которые представляют объектную модель бизнес-процессов. Такие ролевые игры позволяют гарантировать, что каждый прецедент может быть выполнен с использованием имеющихся классов, а значит, объектная модель бизнес-процессов описывает все задачи пользователей. Полезно внимательно проанализировать, что происходит во время такого

моделирования с точки зрения шаблона, показанного на рис. 8.20. Последовательность событий, полученная в процессе ролевой игры, представляет прецедент, сгенерированный на основе сценариев задач. Инициатором этого процесса является руководитель семинара, предлагающий конкретный сценарий. Например, в задаче оформления заказа руководитель, выполняя роль клерка, может сказать: "Элвис, только что позвонил мой приятель и заказал на следующий вторник 200 плиток шоколада".

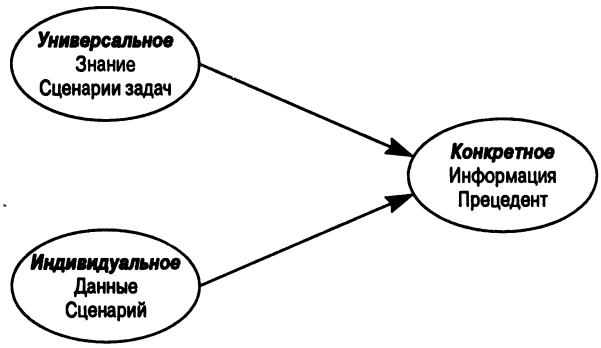


Рис. 8.20. Шаблон силлогизма

Здесь нужно двигаться от индивидуального к конкретному посредством общего: начать с индивидуального сценария, а затем использовать универсальный сценарий задачи, чтобы сгенерировать соответствующий (конкретный) прецедент. Это особый случай силлогизма, хотя читатель должен понимать, что мы используем термин в самом общем смысле Гегеля (Hegel), принятом в [559], а не только в смысле силлогизмов дедукции Аристотеля (Aristotle): "Сократ — человек, все люди смертны, поэтому Сократ смертен". Силлогизмы Гегеля связаны с различными отношениями между категориями, особенно между конкретным, индивидуальным и универсальным. Дедуктивный силлогизм может быть представлен структурой из трех элементов: I-P-U (Individual-Particular-Universal)[9]. В шаблоне силлогизма на рис. 8.20 мы имеем дело с тем же триплетом I-P-U, но там он используется для генерации конкретного (прецедент — последовательность событий) на основе универсального (сценария задачи) и индивидуального (частного сценария). Ясно, что этот подход отличается от обыкновенной дедуктивной логики, но является силлогизмом в понимании Гегеля, т.е. формулой процесса извлечения знания или выявления требований. Это понятие не относится к шаблонам проектирования в смысле работы [291]. Поэтому я не делаю попыток представить его в так называемой форме Александера (Alexander)[10].

Несмотря на то что этот шаблон может показаться трудным для понимания, в действительности он имеет громадное практическое значение как руководство по выделению требований. Кроме того, он подсказывает взаимосвязь между диаграммами последовательностей,

---

[9] Гегель (Hegel) описывает этот силлогизм следующим образом: "Индивидуальность объединяется с универсальностью посредством конкретного". Первый терм — это утверждение о существовании универсального, второй — устанавливает связь индивидуального (экземпляра) с конкретным (классом), что ведет к высказыванию в третьем терме.

[10] Как обсуждалось в главе 7, архитектор Кристофер Александер (Christopher Alexander) распространил среди специалистов по программному обеспечению стандартный способ представления шаблонов [23].

прецедентов и видов деятельности и позволяет показать, что последняя является синтезом первых двух.

8.10. Обеспечение полноты сценариев

Процесс инженерии требований может быть разделен на две части — выделение требований и их анализ. Разработчик выясняет пожелания заказчиков и их потребности, выражая их через агентное взаимодействие и модель задач, а затем строит модель, которая обеспечивает выполнение требований, определенных в более ранних моделях. Теперь ясно, что этот процесс является гладким в том смысле, что объектная модель бизнес-процессов действительно реализует объектную модель задач, а также бизнес-цели и диалоги объектной модели агентов. Это шаг вперед по сравнению с традиционным подходом к "доказательству корректности", который сводится к доказательству того, что модель реализации соответствует спецификации. Однако это все же не гарантирует полноты требований, отраженных в моделях агентов и задачи.

В последнее время появляется интерес к этому вопросу и начинают появляться результаты исследований. Одна из наиболее интересных разработок — это система сценариев для извлечения и проверки правильности знаний в инженерии требований SAVRE (Scenarios for Acquisition and Validation in Requirements Engineering), разработанная как часть проекта CREWS (Cooperative Requirements Engineering With Scenarios), финансируемого Европейским Союзом. В проекте CREWS-SAVRE [512] четко изложены принципы систематизации использования сценариев и генерации полезных сценариев с учетом проблемно-ориентированных шаблонов нормативного поведения и теории человеческих ошибок [650], а также науки о познании и программной инженерии. В работе [636] обращается внимание на то, что существующих методов и средств, таких как Objectory, недостаточно для поддержки принятия решений и управления в этой области. Целью проекта CREWS является систематизация процесса генерации сценариев и критического анализа, повышение их полезности и рентабельности.

Средство SAVRE генерирует сценарии извлечения и проверки требований пользователей почти в автоматическом режиме, основываясь на наборе параметров, установленных пользователем. Затем мастер поддерживает систематический анализ сгенерированных сценариев, гарантируя, что все варианты будут исследованы и все требования проверены. В ходе этого процесса пользователь может добавлять к спецификации обобщенные формулировки требований. Каждый сценарий состоит из гипотетических ситуаций в целевой среде и работает как своего рода сценарий тестирования. В проекте CREWS разработана теория стандартов и альтернатив на основе нормативного поведения, присущего конкретным классам и ненормативным событиям [510]. Пользователь может либо получить пригодный для повторного использования прецедент (сценарий задачи), либо выбрать одну из стандартных объектных моделей системы NATURE (основанную на нормативном поведении объектов в предопределенных прикладных областях), с помощью которой будут сгенерированы сценарии. NATURE [510] — это набор объектных моделей для определенных областей, разработанный как часть другого проекта ESPRIT. В базе данных текущей версии продукта содержится приблизительно 250 типов задач. Сценарий задачи содержит одно исходное событие; по крайней мере, одно из следующих: событие окончания, агенты, поддействия и объекты. Состояние объекта может изменяться в результате выполнения задачи. Задачи связываются с помощью предусловий

действий, определенных в объектной модели. Затем мастер помогает пользователю сгенерировать сценарий с учетом следующих пяти альтернативных направлений.

- Шаблоны взаимодействия агентов
- Типовые исключения (например, действия, которые начинаются, но никогда не заканчиваются)
- Исключения перестановок (например, объединение двух событий, которые происходят одновременно)
- Выбор перестановок (например, временные связи между событиями и действиями)
- Исключения, связанные с решением конкретной задачи (например, отказ устройства)

Например, в системе управления возможен следующий сценарий: "бригада читает мобилизационную инструкцию". Затем система задает ряд вопросов: "Что случится, если событие не происходит?" или "Что случится, если событие происходит дважды?". В ответ пользователь должен добавить примечания к спецификации.

Если мы хотим доказать, что метод SOMA обеспечивает проверку корректности объектной модели по отношению к требованиям, то полнота набора сценариев играет очень важную роль. Возможность проверки соответствия модели требований может оказаться критической для определенных приложений. Кроме того, важную роль играет выделение исключений, связанных с решением конкретной задачи. Однако хранение требований в форме документа, а не в виде объектов приводит к дополнительным ограничениям.

8.11. Множества ассоциаций задачи и диаграммы последовательностей

Когда сталкиваешься с рядом задач или прецедентов, интуитивно понимаешь, что между ними должны существовать связи. Например, библиотечный заказ нельзя зарегистрировать, пока запись о соответствующей книге не появится в базе данных библиотеки. Из этого можно сделать заключение, что задача внесения записи о книге в базу данных позволяет решить задачу выдачи книги по заказу. Мы не можем предоставлять книгу раньше ее закупки, поэтому задача покупки *предшествует* задаче выдачи книги. Читатель, который не в состоянии возвратить книги в срок, попадет в список штрафников и не сможет получить новые книги.

Эти виды взаимосвязей моделируются в методе SOMA с помощью ассоциаций задач. **Ассоциация задачи** (task association) — это именованная, направленная связь между двумя задачами. К числу возможных типов ассоциаций задач относятся: `Succeeds` (Продолжает), `Precedes` (Предшествует), `Enables` (Допускает), `Disables` (Запрещает) и `Parallels` (Выполняется в параллель).

Задачи — это операции объектов-агентов, поэтому ассоциация задачи `Assoc01 (AgentX.A, AgentY.B)` может представлять сообщение, переданное от агента `AgentX` в ходе выполнения задачи `A` к агенту `AgentY`, выполняющему задачу `B`. Агенты могут быть внутренними, внешними или системными. **Набор ассоциаций задачи** (task association set) — это совокупность именованных ассоциаций задачи. Они упорядочены по времени и могут включать параллельно выполняемые задачи с их собственными целями и привилегиями. Это

направленный граф, узлы которого — задачи, а дуги — ассоциации задач. Он представляет связанный, логически последовательный набор задач, которые поддерживают цель бизнес-процесса. Набор ассоциаций задачи эквивалентен одной или нескольким диаграммам последовательностей. Такие диаграммы последовательностей могут включать ветвления и распараллеливания. Простой линейный набор ассоциаций имеет только одну реализацию диаграммы последовательности, тогда как разветвленному набору соответствует по одной диаграмме последовательности для каждого пути в рамках набора ассоциаций. Следовательно, диаграмма последовательностей UML, отражающая лишь подмножество набора ассоциаций, не позволяет выразить полную картину.

Как ясно из главы 6, улучшенная диаграмма последовательностей — мощная методика для анализа и понимания. Ассоциации задач, подобно прецедентам или другим объектам, могут объединяться в любые из четырех структур объектов. Большие наборы ассоциаций задач могут быть разбиты на компоненты, и может быть сгенерирован иерархический набор видов задач. Например, набор ассоциаций, названный "Обработка заказа", может содержать задачи, связанные с проверкой внешней системы, которая выдала заказ, проверкой правильности заказа и т.д.

Процессы, представленные как наборы ассоциаций задач, могут быть многократно использованы во многих контекстах. Хорошим примером является транзакция покупки, структура которой одинакова даже в случае приобретения совершенно различных товаров. Повторное использование задачи такого вида позволяет существенно упростить представление крупномасштабных процессов.

Читатель должен заметить разницу между набором ассоциаций задачи и декомпозицией задачи: все составляющие в композиционной структуре задачи выполняются одним и тем же агентом, а задачи из набора ассоциаций могут выполняться коллективом агентов. Следовательно, набор ассоциаций задачи более близко соответствует объединенному действию в смысле метода Catalysis [204].

В разделе 8.5.1 упоминалось, что диалог между агентами имеет рекурсивную структуру и состоит из шести компонентов. Для представления рекурсии в диалогах, показанной на рис. 8.8, можно использовать диаграммы последовательностей, основанные на наборе ассоциаций задач. Для иллюстрации общей идеи рассмотрим простой процесс покупки чашки кофе. Диалоги, подобно всем прецедентам, могут быть реализованы в диаграммах последовательностей. Диаграмма последовательностей, представляющая такой диалог, показана на рис. 8.21. Набор ассоциаций задачи, отраженный на этой диаграмме, охватывает четыре стадии: *запрос*, *переговоры*, *выполнение* и *переход*. Событие "жажда" заставляет клиента начать процесс запроса чашки кофе (*запрос* или *подготовительная* стадия транзакции). Продавец называет цену, которую клиент взвешивает (*переговоры*) и принимает. Если цена *приемлема, продавец готовит и поставляет кофе* (*выполнение*). Если кофе горячий и достаточно крепкий, клиент берет его и оплачивает (*переход*).

Одним из преимуществ этого способа обоснования диаграмм последовательностей является то, что к модели ассоциаций задач могут быть добавлены различные ветви возможных отказов, что, в принципе, позволяет автоматизировать генерацию диаграмм последовательностей, охватывающих все ситуации.

Процесс может завершиться еще на стадии предложения, например, если кафетерий закрыт или отсутствуют основные ингредиенты приготовления кофе. Переговоры могут потерпеть неудачу из-за того, что цена показалась клиенту слишком высокой. Отказ на стадии выполнения может случиться из-за поломки машины для приготовления кофе. Клиент может

прервать процесс на стадии принятия из-за того, что кофе слишком слабый. В каждом из этих пунктов отказа можно добавить свои ассоциации.

Важным аспектом, который повлиял на обработку диалогов в методе SOMA, является идея вложения диалогов. Любая стадия диалога может включать рекурсивный переход к следующему уровню. Подобная идея заложена и в усовершенствованном методе Catalysis. Набор ассоциаций может включать вложенные наборы для уменьшения сложности модели и управления ею.

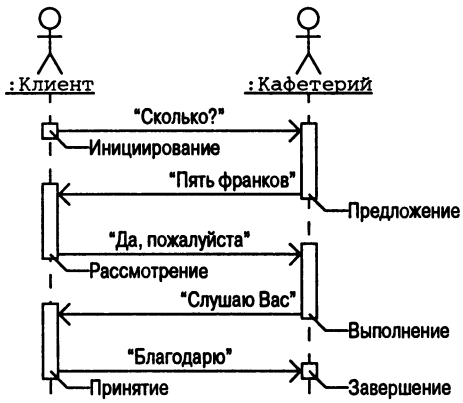


Рис. 8.21. Диаграмма последовательностей для представления диалога

Описанный подход несколько иначе реализован в программном продукте, созданном компанией Action Technologies Inc. (ATI). Он дает возможность специалистам по моделированию формировать диалоги по схеме, показанной на рис. 8.7, и генерировать потоки данных в таких продуктах, как Lotus Notes. В результате работы Винограда (Winograd) и Флореса (Flores) подверглись критике со стороны специалистов в области инженерии требований как ведущие к негибким, жестким режимам, которые не любят пользователи. Мне кажется, что это не уменьшает значения основной теории, поскольку наша цель состоит не в постройке модели технологического процесса, а в том, чтобы обеспечить исходную точку для объектно-ориентированного анализа.

Диаграммы последовательностей UML

Явления, которые происходят в указанной последовательности, но не отражают семантической связи, описываются диаграммами последовательностей UML. Это важно понимать. Стрелки на диаграмме последовательностей UML представляют последовательность действий и ничего больше.

При помощи диаграммы последовательностей UML легко представить следующую последовательность действий.

1. Солнце всходит
2. Птицы начинают петь
3. Движение в часы пик приводит к пробкам

Здравый смысл позволяет заключить, что первое и второе явления причинно связаны (в некотором смысле, солнце посылает электромагнитное сообщение птицам, которое заставляет их петь). Причинные связи между первыми двумя шагами и третьим менее ясны. Конечно, пение птиц никак не влияет на изменение условий движения. Восход Солнца тоже имеет малый прямой эффект. Хотя большинство из нас пользуются электрическими лампочками, тем не менее мы обычно ходим на работу в течение светового дня, и в этом смысле восход Солнца будет предвещать перебои в движении. Однако для выяснения этой связи необходимо расширить нашу модель путем включения некоторых категорий людей, аспектов землепользования, моделей движения и транспортных моделей. В этом заключается сила и слабость диаграмм последовательностей UML. С одной стороны, они позволяют обойти ненужную, по существу, стадию смыслового связывания набора действий, что упрощает построение диаграммы. С другой стороны, рассмотрение незавершенной диаграммы может привести к ошибочным предположениям относительно причинной связи между ее объектами. Это становится особенно важным при передаче спецификации от аналитика к разработчику. В конце концов программа должна быть написана, она должна быть полной и последовательной. Без четких рекомендаций разработчик может только сделать предположения о последовательности событий в потоке управления. Для устранения этой проблемы необходимо игнорировать время и давление бюджета, что существует в реальных организациях. Ошибки в проектном решении приводят к несогласованности потока управления программы. Они могут превратиться в дефекты, обнаружение которых обойдется очень дорого, особенно если этот вид ошибки не прослеживается от спецификации. Это одно из проявлений хорошо известной взаимосвязи процессов моделирования: время, которое экономится на этапе анализа, ведет к удорожанию стадии поддержки.

Пример

В учебном материале, поставляемом с версией UML от компании Rational Rose, мы нашли утверждение, что диаграммы последовательностей можно использовать на ранних этапах анализа требований. В качестве примера приводится относительно простой процесс обработки заказа. Естественно, на диаграмме последовательностей появляются клиент и клерк. Клерку необходимы сведения о наличии товара и о его цене, поэтому он ищет их в прейскуранте и в книге учета запасов. Здесь возникает соблазн разместить на диаграмме экземпляры `stock_record` (книга запасов) и `price_list_item` (прайс-лист). Компания Rose пытается упростить ситуацию, вводя классы `StockRecords` и `PriceList`. Любому опытному разработчику объектных систем очевидна ошибочность такого подхода. Ему ясно, что класс `Product` (Товар) должен знать (или иметь возможность узнать) объем запасов и цену любого своего экземпляра или хотя бы средние величины. Следовательно, разработка диаграммы последовательностей до построения основной модели классов очень опасна.

Таким образом, предложенный компанией Rational Rose метод может привести к абсурдно плохому проекту с точки зрения объектной ориентации. Мы рекомендуем другой метод, связанный с построением объектной модели бизнес-процессов, основанной на объектной модели задачи. Разработчики должны знать, что класс `Product` должен обладать информацией о цене продукта и уровне запасов еще до построения диаграммы последовательностей. Эти диаграммы должны использоваться для документирования и проверки сложности бизнес-процессов после того, как объектная модель построена.

Еще одна проблема, связанная с диаграммами последовательностей UML, состоит в том, что их элементы интерпретируются как экземпляры классов системы, посылающие друг

другу сообщения, представленные горизонтальными линиями со стрелками. В методе Catalysis эти стрелки рассматриваются как прецеденты, что является лучшей интерпретацией. Вертикальные прямоугольники на диаграммах, подобные показанным на рис. 8.21, представляют действия. В нашем методе экземпляры рассматриваются как агенты реального мира, а прямоугольники представляют их задачи. Горизонтальные линии со стрелками представляют ассоциации задачи, такие как предшествование. Я думаю, что это лучший метод в контексте инженерии требований, потому что он приближает модель к реальным условиям, привычным для пользователя. Это ведет к более плодотворному диалогу между пользователями и разработчиками во время анализа требований и улучшает общее понимание проблемы. Интерпретация метода Catalysis может использоваться и в спецификации.

8.11.1. Конъюнктивные, дизъюнктивные и вложенные наборы ассоциаций

Мы уже видели, что наборы ассоциаций задач могут включать параллельные ветви. В качестве примера рассмотрим процесс приготовления чашки растворимого кофе. Он включает наполнение чайника водой и доведения ее до кипения. Пока греется вода, в чашку добавляются (по выбору) молоко и сахар. После того как вода в чайнике закипит, гранулы кофе должны быть растворены в горячей воде. Эти задачи и их взаимосвязи подразумевают набор ассоциаций с двумя параллельными действиями. Эту информацию трудно показать на диаграмме последовательностей UML. Конечно, параллельные ветви могут быть представлены на отдельных диаграммах последовательностей.

Иногда представление наборов ассоциаций задачи на диаграммах последовательности может быть довольно громоздким. В таких случаях для более ясного понимания структуры можно выполнить декомпозицию набора ассоциаций, используя идею уточнения из метода Catalysis. Отдельную сложную задачу можно заменить набором ассоциаций или вложенной структурой. На рис. 8.22 `ассоциация1` связывает задачу `задача1` с полным набором ассоциаций `набор задач 1`. Подобным образом `ассоциация2` и `ассоциация3` связывают задачу `задача3` с полным набором ассоциаций. При развертывании набора ассоциаций (щелчком кнопкой мыши на его изображении) появляется диаграмма для вложенного набора ассоциаций. Это показано на рис. 8.23 для набора ассоциаций `ассоциация3`.

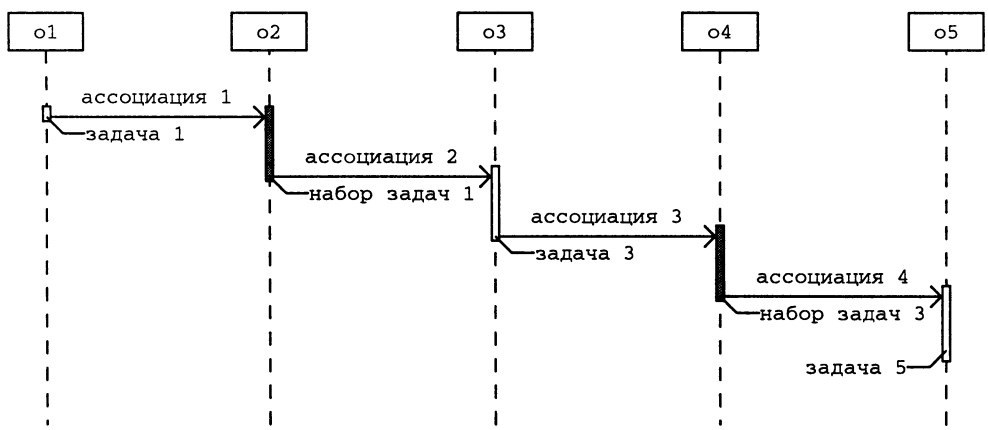


Рис. 8.22. Набор ассоциаций для вложенных задач

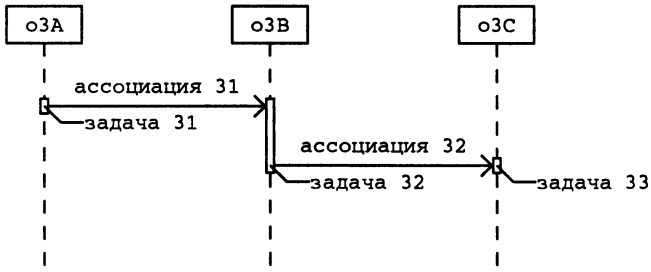


Рис. 8.23. Набор ассоциаций для одной задачи

Создание набора ассоциаций

Наборы ассоциаций являются объектами и могут быть связаны всеми четырьмя структурными взаимосвязями, включая композицию. В приведенном примере наборы задач 1 и 3 вложены в набор ассоциаций, названный `Main` (т.е. являются компонентами). Структура композиции для `Main` показана на рис. 8.24.

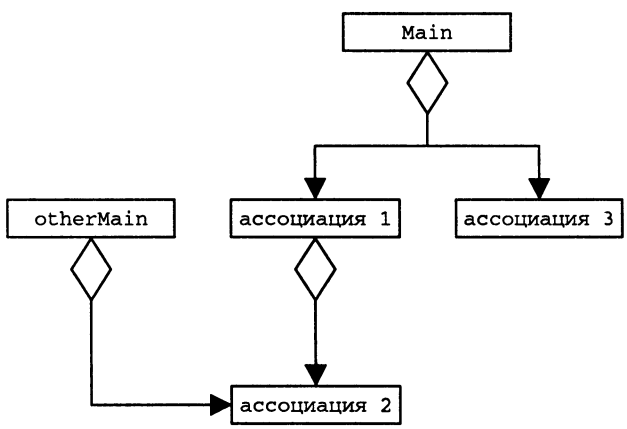


Рис. 8.24. Структура композиции для набора ассоциаций

Развертывание набора задач позволяет увидеть, что `набор задач 2`, в свою очередь, является компонентом другого набора ассоциаций, названного `otherMain`. Заметим, что основой предлагаемого метода является идея повторного использования спецификации.

8.12. Выполняемые спецификации и моделирование

Главным преимуществом метода SOMA и некоторых других методов, таких как ROOM [694], является возможность *реализовать спецификацию*. Персонал может испытывать значительные трудности в понимании динамики системы, если в спецификации заложено только

статическое представление. Поэтому хорошие средства поддержки должны обеспечить возможность задания сценариев на языке высокого уровня, описывающих взаимодействие классов и агентов в динамике. Таким языком может быть подмножество языка Java с добавлением некоторых дополнительных возможностей. Это позволит протестировать каждую задачу, выделенную на ранних стадиях моделирования и анализа бизнес-процессов, а также классы и операции, определенные в процессе разработки технических требований. Каждой операции можно поставить в соответствие сценарий, который может служить ее спецификацией. Для описания операций лучше использовать язык *сценариев*, а не язык *программирования*. Это облегчит задачу проектировщика, которому не придется строго выполнять правила синтаксиса сложного языка.

Напомним, что сообщение в модели бизнес-процесса представляет диалог, а сценарий задачи определяет аспект выполнения соглашения, неявно описываемого этим диалогом. Если на этапе моделирования задача выполняется в результате некоторого события, определяемого в объектной модели агента AOM (Agent Object Model), то мы последовательно переходим от этой модели к объектной модели задачи TOM (Task Object Model), а затем к объектной модели бизнес-процессов BOM (Business Object Model). Важно четко представлять значения этих переходов. AOM и TOM относятся к среде пользователей и их бизнес-процессов. Эта часть модели в [188] названа **основной**. Эти модели связаны с видами деятельности, выполняемыми в том случае, если предлагаемая система будет использована для бизнеса. Модель BOM относится к системе. В работе [188] она названа **моделью спецификации**. Эта модель определяет классы, которые должны быть созданы. Цель моделирования состоит в подтверждении того факта, что классы BOM на самом деле позволяют решать задачи пользователя, определенные в процессе анализа.

Для проверки правильности проектного решения каждого класса нужно связать эти три модели воедино и протестировать, как каждый класс поддерживает выполнение задач агентов. Агент при моделировании бизнес-процесса знает о реализации событий, инициирующих диалоги, и поддерживает их.

Сообщения всегда имеют одну корневую задачу, которая может делиться на несколько составляющих. Одна и та же задача может быть использована (или повторно использована) одним из агентов в нескольких диалогах. Объекты и классы задач (т.е. бизнес-объекты в BOM) не обязательно связаны друг с другом соотношением 1:1. Отображение может быть более сложным. Важно помнить, что каждую корневую задачу поддерживает только один инициирующий класс, как это мы уже видели в разделе 8.8. Хотя при выполнении некоторой задачи несколько классов могут взаимодействовать друг с другом, но только один из них является инициатором этого взаимодействия. Выполнение задачи всегда начинается с выполнения определенной операции. Эта отправная точка записывается в задаче. Она обеспечивает связь области определения класса с областью определения задачи: задача **выполняется** некоторым классом с помощью определенной операции.

Если главные части бизнес-процесса промоделированы правильно, то диаграммы последовательностей можно использовать для создания регрессионных тестов, периодически выполняемых на определенных этапах разработки и позволяющих гарантировать, что модификация объектной модели бизнес-процессов не искажает понимания проблемы и не приводит к отклонению от основной задачи. Результирующая диаграмма последовательностей может быть использована для подтверждения правильности проектного решения класса и для регрессионного тестирования.

8.12.1. Дискретные события и моделирование во времени

Моделирование составляет основу объектной технологии с первых дней развития объектно-ориентированного программирования и языка Simula. Язык Simula появился в 1949 году как результат работы по моделированию дискретных событий и исследования операций в области проектирования ядерных реакторов. Дифференциальные уравнения, используемые для решения таких задач, очень сложны даже для численного решения, поэтому с самого начала разрабатывалась альтернативная методика. Она заключалась в моделировании мира, который находится в динамике и изменяет свое состояние дискретно в соответствии с определенными временными интервалами. При таком моделировании, которое можно многократно повторять на компьютере, можно раньше наблюдать результаты влияния различных параметров, чем получать расчетные данные. Это аналогично использованию электронных таблиц для анализа "что..., если...". Однако языки, которые существовали в 1960-е годы, не особенно подходили для моделирования. Абстракции традиционных языков, подобных FORTRAN, не подходят для компьютерного описания процессов. Для описания процессов реального мира более пригоден язык Simula.

Изначально в модель входили такие объекты, как нейтроны и поглощающие стержни ядерного реактора. Более понятный пример — моделирование дорожного движения на сложных участках, которое включает оценку частоты наступления критической ситуации, оценку насыщения потока, управление светофорами и формирование очередей. Система через дискретные промежутки времени отслеживает ситуацию, оценивает сложившуюся ситуацию на каждом временном шаге в терминах длины очереди, времени задержки и т.п. Существует возможность наблюдения глобальных явлений, оценки устойчивости или сходимости к устойчивому состоянию. Построенная модель системы может быть протестирована на соответствие различным предположениям.

Вид моделирования, обсуждаемый так подробно в этом разделе, базируется на событиях, вызывающих передачу сообщений между агентами, которые, в свою очередь, инициируют операции бизнес-объектов. Моделирование дискретных событий обычно происходит в дискретном времени.

Свойства объекта задачи должны включать ограничения, относящиеся к производительности системы.

- Среднее время выполнения задачи
- Максимальное и минимальное время выполнения задачи
- Среднеквадратичное отклонение времени выполнения
- Распределение вероятностей времени должно удовлетворять одному из стандартных законов распределения: нормальному, логарифмическому, Пуассона, экспоненциальному, равномерному и т.д.
- Число очередей и доступных серверов очередей
- Используемые единицы измерения времени

Эта спецификация включает распределение вероятностей, представляющих способы реализации событий из очереди. Если в модели порядок реализации событий из очереди не определен, то обычно выбирается распределение Пуассона. Общеизвестно, что пуассоновскими процессами описывается трафик в компьютерных сетях и движение транспортных средств.

Если в модели представлено время, необходимое для выполнения некоторой задачи вручную, например для заполнения налоговой декларации, то для его описания больше подойдет нормальный закон распределения.

Заметим, что спецификация выполнения задачи включает число очередей и серверов очередей. Например, в зале регистрации авиакомпании зачастую имеется несколько стоек обслуживания, к каждой из которых стоит отдельная очередь. Иногда при наличии нескольких серверов информация обрабатывается из одной очереди.

Моделирование дискретных или временных событий дает возможность проектировщикам наблюдать процесс разработки системы по времени, но время при этом моделируется как последовательность дискретных равномерно расположенных моментов.

Многопоточность

В реальности бизнес-процессы обычно выполняются параллельно. Это повышает сложность моделирования процессов из предметной области. Может возникнуть необходимость синхронизировать потоки в различных точках, что еще более усложняет жизнь.

Инструментальные средства моделирования должны обеспечивать поддержку для моделирования параллельных потоков. Межпроцессорную синхронизацию можно осуществить при помощи модели семафора или подобной модели, в которой одновременно выполняемые действия могут порождаться любым действием любого класса. Создается видимость, что все действия, определенные внутри потоков, порождаются сразу и параллельно. Конечно, в большинстве случаев моделирование может быть выполнено на обычной последовательной машине или однопоточной SISD-машине (Single Instruction stream, Single Data stream). Это означает, что на самом деле процессы не выполняются параллельно, а лишь создается видимость многопоточности. При этом потребуется переменная, которая будет отвечать другим экземплярам процесса и передавать уведомления об изменениях состояния, возможно, используя шаблон Observer, описанный в главе 7. Такие уведомления могут порождаться любой операцией любого класса, в операционный сценарий которого включены соответствующие директивы. Нужно иметь возможность устанавливать и сбрасывать семафоры.

Должна быть предусмотрена возможность приостановки выполнения процесса. Промежуток времени, в течение которого процесс "спит", может быть фиксированным или вычисляемым. В частности, процесс может "спать", пока не будет установлен один или несколько семафоров.

Событийное и временное моделирование могут быть объединены. Одной из основных причин такого объединения может быть создание фоновых процессов для моделирования дискретных событий. Это можно реализовать за счет применения событийного моделирования поверх временного.

8.13. Требования к организации и проведению семинаров

В этой главе несколько раз упоминалось о совместных семинарах пользователей с разработчиками. Рассмотрим детали организации этих семинаров, поскольку они являются важным компонентом процесса SOMA. Существует несколько формальных методов структурирования семинаров, но эта глава содержит ряд рекомендаций, основанных на опыте проведения семинаров за последние два десятилетия.

Акцент в этом разделе делается на организации семинаров и управлении их ходом для успешного достижения поставленных целей. Макропроцесс разработки, в рамках которого реализуется микропроцесс семинара, будет рассматриваться в главе 9.

Исторически сложилось, что технические требования к системам определяются во время опроса пользователей. Системные аналитики опрашивают отдельных пользователей, а иногда и небольшие группы обо всех аспектах системы. Результаты этих опросов собираются в отчеты по системному анализу, которые затем распространяются среди участников проекта. На основе полученных откликов выполняется пересмотр требований и т.д. Такие документы обычно очень громоздки и зачастую трудно читаемы. Трудно поверить, что автор когда-то сам читал их и все понимал. Можно предположить, что эти "опусы" составлялись из-за отсутствия полного понимания, а не для того, чтобы спровоцировать напрасное противостояние.

В результате этого возникает несколько проблем.

- Каждый опрашиваемый пользователь склонен рассматривать систему со своей точки зрения. Это естественно и правильно, но отсутствие мнений других пользователей системы не делает систему более прозрачной.
- При таком подходе возникает барьер между пользователями и разработчиками, а иногда между различными группами пользователей. Пользователи задают вопросы об особенностях реализации требований. Разработчики уходят от ответа. Результат обычно не подходит пользователю, возможно из-за того, что требования к системе неудачно воплощены. За этим неминуемо следует порицание и выявление виновных.
- Организация полного опроса с учетом свободного времени каждого пользователя может оказаться очень длинным процессом. Только первичный опрос может занять два или три месяца, особенно если пользователи географически удалены друг от друга. Часто необходим второй этап опроса для уточнения или устранения противоречий. Все это увеличивает время на разработку общего проекта, а в изменяющейся окружающей среде вызовет проблемы с внедрением системы. Требования пользователя, высказанные в начале процесса разработки, неизбежно изменятся, так как изменится среда. Чем дольше длится процесс разработки, тем выше вероятность несоответствия системы изменившимся требованиям. Это явление называют "смещением" требований.

В противоположность этому семинары обеспечивают следующие преимущества.

- Снижают время, необходимое для установления требований.
- Обеспечивают прямое обсуждение требований с пользователями и облегчают возможность достижения компромисса.
- Разработчики получают информацию от пользователей из первых рук.
- И разработчики, и пользователи чувствуют себя причастными к проекту.
- Зачастую (но не всегда) сокращаются денежные и другие затраты на выявление требований и процесс их обработки.
- Устанавливается хороший темп для быстрого развития процесса. Прозрачность зарегистрированных технических требований, выработанных совместными усилиями в течение нескольких дней обсуждений, приводит к повышению оптимизма по поводу проекта.

8.13.1. Распределение ролей во время семинаров

Участниками семинаров являются спонсоры, пользователи, специалисты по предметной области, руководитель семинара, разработчики технических требований, руководитель проекта и члены группы аудиторской проверки. Кроме того, семинары могут посещать наблюдатели (у которых есть возможность пройти отличное обучение), которые, имея определенный опыт, могут повлиять на проектное решение.

Разработчики требований — это опытные системные аналитики, которые должны понимать метод SOMA и принципы объектного моделирования. Они документируют требования во время семинаров. Для выполнения этой задачи необходим очень высокий уровень квалификации. На больших семинарах эту функцию могут выполнять два человека.

Спонсор — это основной исполнитель, отвечающий за рассматриваемую предметную область. Его назначение состоит в том, чтобы оценить результаты семинаров и стоимость проекта, решить проблемы, которые не могут быть решены непосредственно по ходу дела, и гарантировать выбор соответствующих участников. Спонсор должен посещать обзорную часть семинара, но не должен посещать детальные.

В основную бригаду входят разработчики и от двух до двадцати основных пользователей, являющихся экспертами в анализируемой области. Их роль состоит в том, чтобы выявить бизнес-требования и проверить, чтобы эти требования были правильно записаны.

Руководитель семинара — один из главных участников, определяющих успех семинара. Это опытный бизнес-аналитик, выполняющий следующие функции.

- Проводит детальные и обзорные заседания
- Гарантирует, что требования фиксируются в полном объеме и не противоречат друг другу
- Обеспечивает нужный уровень детализации для дальнейшей работы над проектом

8.13.2. Кто должен посещать семинары

Не каждый сотрудник организации может позволить себе посещать семинары по определению требований. Типичным примером являются сотрудники бухгалтерии, которые отказываются оставлять свои рабочие места. Следовательно, некоторые пользователи выступают в роли делегатов от имени своих непосредственных коллег или менеджеров. Выбор делегатов играет определяющую роль в достижении успеха семинара, поэтому среди участников должны быть представители как пользователей системы, так и команды разработчиков.

Пользователи

Присутствие основных пользователей является чрезвычайно важным моментом. Конечно, следует отделять подлинных пользователей системы от сторонних сотрудников. При этом нужно учитывать несколько факторов.

- Важным фактором является опыт работы. Люди с большим опытом могут лучше понимать проблемы, рассматриваемые на семинарах. Однако молодые сотрудники лучше разбираются в деталях бизнес-процесса. Поэтому нам нужны люди с различной выслугой лет — прежде всего, мы нуждаемся в знающих людях, которые не боятся высказывать собственную точку зрения, возможно, отличную от мнения руководителя. Вот почему так важна роль руководителя семинара.

- Каждый представитель заинтересованных лиц *должен* иметь полномочия подписывать результаты семинаров.
- Сложность взаимодействия экспоненциально возрастает по мере увеличения числа участников. Однако на семинаре должны быть представлены все группы участников проекта. Если в системе будет выявлен недостаток, то за него в равной степени должны отвечать пользователи и заинтересованные лица.

Перед началом семинара спонсор, руководитель проекта и руководитель семинара должны предварительно встретиться, чтобы утвердить список участников. Они должны рассмотреть всех претендентов, исследовать предварительный список, попытаться откорректировать его и создать документ с заключительным списком для рассылки приглашений. Можно было бы разработать матрицу кандидатов, перечисляя всех людей, которые могли бы посещать семинар, а затем сравнить возможные комбинации кандидатов из матрицы с точки зрения успеха семинара. Начать нужно с диаграммы организационной структуры компании. Строки содержат организационные единицы, колонки — приблизительные уровни иерархии внутри организации, а ячейки — названия логических заданий и возможных кандидатов на их выполнение. Надо заметить, что это *не* формальный документ, так как логические роли могут быть предметом обсуждения и изменяться во время семинара.

Например, рассмотрим систему поддержки нового процесса представления товаров, продажи, оформления заказов и производства. Она затрагивает компетенцию отделов продаж, маркетинга и производства. Возможная матрица участия в семинаре показана на рис. 8.25. Нужно удостовериться, что матрица отражает фактическую организационную структуру, а затем указать имена кандидатов для участия.

	Стратегический менеджмент	*Тактический менеджмент*	*Оперативный и организационный менеджмент*
Продажи	Директор по продажам Региональный менеджер по продажам	Бухгалтеры Региональные менеджеры	Персонал по сбыту Персонал по обслуживанию и телепродажам
Маркетинг	Директор по маркетингу	Менеджеры по производству	Ассистенты по маркетингу
Производство	Директор по сбыту	Инженеры-технологи	Управляющие складом Механики

Рис. 8.25. Матрица участников семинара

Пользователи должны посещать все заседания семинара. Это проще потребовать, чем выполнить. Отвлечение людей от важной работы, даже в течение нескольких дней, может иметь значительные последствия. Если пользователь уделит семинару пару часов, а затем на некоторое время исчезнет, то это может пагубно сказаться на результатах семинара. Поэтому

спонсор и руководитель проекта должны поработать с менеджерами, чтобы гарантировать успешное проведение семинаров. Семинар не должен представлять собой набор совещаний с высокой текучестью участников.

Разработчики

Если это возможно, семинар должна посетить вся группа разработчиков. Основная идея метода SOMA — создание *объединенной* группы из пользователей и разработчиков. Причем необходимо избежать разбивки на "нас и их", когда группа пользователей выдвигает требования, а группа разработчиков создает систему по своему усмотрению, не считаясь с потребностями пользователей. В этом случае пользователи не смогут сразу сообщить разработчикам, где они пошли по неправильному пути. Зачастую ошибки выявляются довольно поздно, когда уже сложно переделать готовый продукт. Если на семинаре присутствует вся группа разработчиков, то ее члены смогут лучше осознать системные требования. Важно, чтобы все разработчики чувствовали все части проекта.

8.13.3. Выбор места проведения семинара

Главный вопрос, который стоит перед организатором семинаров по разработке технических требований, сводится к выбору места проведения совещания. Семинар можно проводить на месте разработки или по месту работы пользователей. Доводы в пользу обеих стратегий приводятся в табл. 8.3.

Таблица 8.3. Сравнение стратегий проведения семинаров а месте разработки и за его пределами

	На месте	За пределами
За	Проще убедить пользователей и специалистов по предметной области посещать семинары	Смена обстановки часто расслабляет участников, особенно если обычно они напряженно работают
	Проведение семинара на месте обычно обходится дешевле	Меньше возможностей увильнуть для пользователей и специалистов по предметной области
	Снижаются транспортные издержки	Увеличиваются возможности управления семинаром
Против	Семинар может постоянно прерываться, и пользователи могут часто отзываться для выполнения другой работы. Это наиболее важная и общая причина неудачи семинаров	Затраты на размещение обычно выше
	Пользователи будут думать о своих традиционных обязанностях и не смогут сосредоточиться на проблемах семинара	Значительные дополнительные транспортные издержки

Наиболее обычное место проведения семинара — гостиница. Выберите достаточно комфортную гостиницу, расположенную не очень близко к рабочим местам. Если фирма по разработке программного обеспечения имеет достаточное количество помещений, то семинар можно провести на ее территории.

8.13.4. Вопросы логистики

Нужно подготовить основное и дополнительные помещения. Они могут быть очень маленькими, в которых будут только стол, кресла и плакаты. Если семинар большой, необходимо несколько дополнительных помещений. Дополнительное помещение может потребоваться, если необходимо выделить подгруппу, которая должна решать некоторый важный, но не обязательно основной вопрос. Кофе и завтрак могут быть организованы в дополнительном помещении, но в любом случае это должно быть за пределами основного пространства. Это гарантирует, что перерыв действительно будет перерывом, когда люди смогут расслабиться или немного отвлечься в другой обстановке. Очень важно, чтобы кофе не отвлекал внимания участников семинара во время заседаний.

Планировка помещений тоже важна. На рис. 8.26 показана типичная, но не обязательно идеальная планировка. Например, во многих случаях допускается, чтобы кофе подавался в основном помещении, но это плохое решение, так как это может отвлекать слушателей. С другой стороны, если нет никакого другого помещения, то кофе "в зале заседаний" — это лучше, чем отсутствие кофе. Все участники должны иметь тесный контакт. Это важно. Поэтому лучшая планировка — это подковообразный стол. Отдельные столы вообще не должны использоваться, поскольку в этом случае участники семинара могут разбиться на группы, чего не следует допускать.

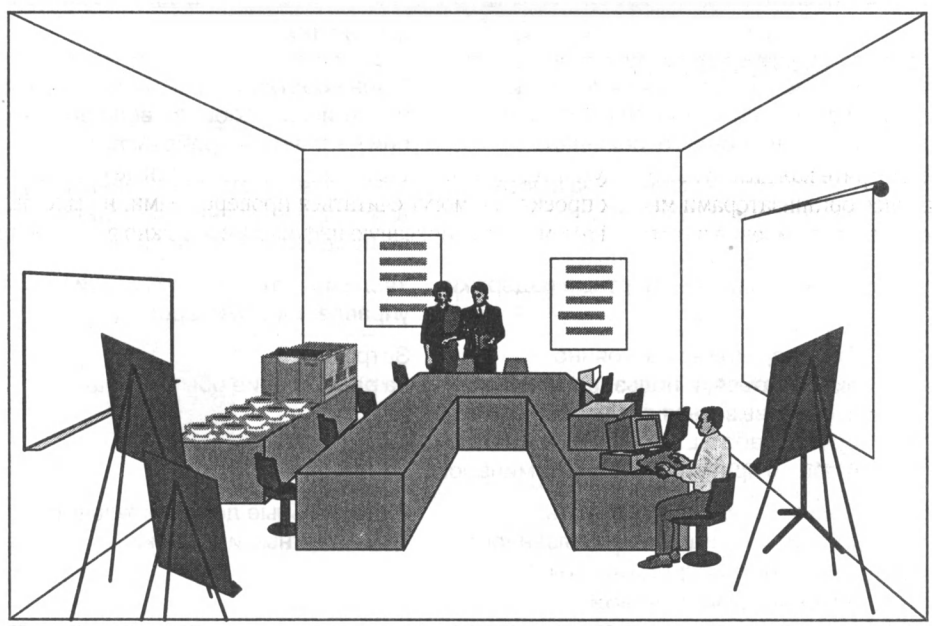


Рис. 8.26. Типичная планировка комнаты проведения семинаров

Другие столы, показанные на рисунке, используются для установки принтера и другой техники. Полезна также доска. Необходимо запастись маркерами, проверить, что все они работают, а также удостовериться, что они стираемы или растворимы в воде. На рис. 8.26 показана обычная белая доска (whiteboard) и проекции диаграмм на стене.

За компьютером (справа) находится специалист по инженерии требований. Компьютер оборудован любым CASE-средством или другими необходимыми инструментальными средствами, в том числе и стандартным набором программного обеспечения автоматизации делопроизводства типа MS Office. Этот компьютер соединяется с быстрым лазерным принтером и проектором.

На рисунке не показаны некоторые устройства, в том числе видео- или звуковое оборудование, которое желательно тоже использовать.

Необходимо убедиться, что гостиница может обеспечить адекватные средства фотокопирования. Во время перерыва текущие проектные документы должны быть напечатаны и фотокопированы, чтобы после перерыва участники имели перед глазами текущие документы. Необходимо удостовериться, что руководство гостиницы ясно понимает, что вам ежедневно нужно снимать 20 копий документов по 30 страниц каждый. Если оно не может выполнить эти требования, найдите другой выход.

Установите и протестируйте оборудование накануне вечером. Это особенно необходимо, если вы арендуете оборудование. Проверьте все: *установлен ли драйвер принтера*, *есть ли бумага и тонер для принтера*, *есть ли кабель для проектора*.

При создании карточек классов можно разбить пользователей на две бригады, а затем объединить полученные карточки на пленарном совещании. Может показаться, что этот процесс занимает больше времени, но часто он позволяет выявить важные конфликты и дополнительные варианты. Следует учитывать, что маленькие группы обычно работают более результативно, чем большие.

8.13.5. Контрольные списки

Приведенные ниже контрольные списки предназначены для организаторов и руководителя семинара. Они должны восприниматься как рекомендации, а не руководство к действию. Рассмотрите каждый пункт и подумайте, *нужно ли вам это*. Эти контрольные списки использовались организаторами многих проектов и могут считаться проверенными; но мне бы хотелось услышать мнение читателей о том, какую полезную информацию можно в них добавить.

Первые два контрольных списка охватывают повестку дня обзорного и детального семинаров соответственно. Третий список, возможно, наиболее полезен. Это список полезных или необходимых дел, которые необходимо выполнить перед началом семинаров.

Первый контрольный список.
Вопросы, обсуждаемые на обзорных заседаниях

```
Введение (необязательно)
Формулировка миссии
Задачи
Метрики
Выявление приоритетов
Обсуждение приоритетов
Внешняя модель процесса
  Внешние агенты и диалоги
```

Внутренняя модель процесса
  Агенты и исполнители
  Диалоги
  Триггеры и цели
  Ключевые задачи
Исключения
Допущения
Основные вопросы — кто будет их решать?
Приоритеты реализации
Возможности повторного использования классов
Следующие шаги
Подписание документов пользователями

Второй контрольный список.
Вопросы, обсуждаемые на заседаниях, посвященных детальному анализу

Введение (если необходимо)
Результаты обзорного семинара (если есть в наличии)
  Миссия, цели, метрики и приоритеты
  Моделирование бизнес-процесса
Моделирование процесса
  Анализ каждого диалога
  Цель - Предложение / Запрос - Переговоры - Корневая задача - Передача
  Триггеры для агентов
  Задачи верхнего уровня (корневые) для каждого сообщения
Исследование возможностей реорганизации бизнес-процесса
  Существуют ли диалоги для каждой цели?
  Существует ли по крайней мере одна задача для каждого из диалогов?
Разложение сценариев задач на подсценарии до атомарного уровня
Исключения: побочные сценарии
Совместная работа
  Текстовый анализ для поиска предполагаемых классов
  Карточки классов
  Структуры (классификации, композиции, ассоциации и использования)
  Сортировка карточек для уточнения структуры
  Задание атрибутов классов
  Задание операций классов
  Задание правил классов
  Модели состояний при необходимости
Обобщение результатов работы
Подтверждение ассоциаций - необязательно; десять минут максимум
Ролевые игры - диаграммы последовательностей для каждой корневой задачи
Анализ библиотеки классов
Обсуждение приоритетов реализации
Повторное использование
Решены ли проблемы, выявленные на обзорном семинаре?

```
Рассмотрены ли операционные аспекты?
Рассмотрены ли юридические аспекты?
Построена ли политика безопасности?
Построена ли политика восстановления информации?
Подписание документов
```

Третий контрольный список.
Вопросы логистики

```
Заказать гостиницу/помещение
Определить число участников
  Помещение достаточно большое?
  Достаточно ли вспомогательных помещений?
Определите планировку помещений
Вопросы питания
  Расписание обедов и завтраков
  Кофе за пределами основного помещения
Мероприятия по фотокопированию
Информирвоать участников, помощников, аналитиков
  Подтвердили ли они свое согласие на участие?
  Мероприятия по путешествиям — заказаны ли билеты?
  Информированы ли помощники?
Заказать оборудование, установить и проверить программное
обеспечение
  Компьютер, принтер, драйверы, дисплеи, проектор
  Установлено и проверено ли программное обеспечение?
      MS Office, Visio или что-то подобное
      Другое программное обеспечение для
      автоматизированного проектирования и разработки
      Существует ли библиотека классов и карточки классов?
      Достаточно ли дискового пространства?
      Установлен ли необходимый драйвер принтера?
Установлено ли аудио/видеооборудование?
Есть ли плакаты, ручки, печатные бланки, аудио/видеозаписи,
бумага для принтеров и участников, карандаши, бумага для заме-
ток,
кнопки, цветные наклейки и т.д.?
Есть ли запасные лампочки, батарейки и т.д.?
Есть ли доска?
Соберите и возьмите документы предыдущих семинаров
Соберите описания классов и задач из более ранних семинаров
Подготовьте повестку дня
Подготовьте документы для подписи
```

8.13.6. Требования к помощникам руководителя семинара

Помощники руководителя семинара должны обладать разносторонними навыками. Они должны быть хорошо знакомы с методами разработки и языками моделирования: UML, Catalysis и SOMA. Они должны быть настойчивы и знать, когда необходимо быть на заднем плане и только слушать. Они должны уметь поощрять застенчивых участников и научить их себя вести. Очень полезно установить основные правила проведения семинаров.

Легкий юмор располагает людей к непринужденной обстановке, и помощник не должен бояться тонких и даже рискованных замечаний. Ключевое умение состоит в способности понять главное в процессе дискуссий и быстро увидеть основные противоречия. Автор в процессе анализа обычно выделяет два противоположных направления, а затем прослеживает связь между ними. Этому можно быстро научиться. Единственный способ стать хорошим помощником — постоянно учиться. Возможно, некоторые люди рождаются помощниками. Автор же учился на практике.

Важно, чтобы помощник соблюдал нейтралитет по отношению ко всем группам участников. Не должно быть никакого предпочтения никакой группе пользователей или разработчиков. Это важно, потому что иногда помощником является сотрудник организации-разработчика.

Голосование — важный способ достижения согласия, но эта процедура должна быть тщательно отработана.

Руководитель семинара отвечает за регламент проведения заседаний и должен отслеживать время каждого мероприятия в повестке дня. Он должен четко понимать, где возможны задержки. Помощник должен разъяснять возникающие проблемы и замедлять темп, когда это необходимо. Между руководителем семинара и его помощником должна существовать глубокая связь, возможно даже на уровне телепатии.

Недостаточно хороший помощник характеризуется любыми из следующих качеств: хвастливость или деспотизм, робость, недостаток доверия, идея фикс, неспособность сидеть и слушать, лесть по отношению к руководству, недостаток осведомленности об общих и конкретных проблемах бизнес-процессов или недостаток обаяния. Любопытно, что эти недостатки проявляются только в работе.

8.13.7. Кто должен вести записи

Протоколы семинаров ведет инженер по требованиям (requirements engineer). Он должен владеть соответствующим программным обеспечением и бегло и точно набирать текст. Но это еще не все. Инженер должен разбираться в записях стенографистки (при необходимости) и уметь представить диаграммы в виде связного текста.

Иногда полезно записывать заседания семинара с помощью аудио- или видеотехники. Наличие записанного на пленку отчета главным образом важно для аналитиков и тех, кто не сумел понять или записать некоторый нюанс. Видеоинформация позволяет наглядно восстановить картину обсуждения, поэтому ее более высокая стоимость может быть оправдана. Аудио- и видеозаписи не стоит переводить в машинописный текст. Никто не читает такие документы, а процесс перевода трудоемок и дорог. Видео- или аудиозаписи можно отправить каждому участнику семинара. При этом участники семинара должны помнить о конфиденциальности информации.

8.13.8. Проведение семинара

Семинар SOMA может длиться от одного до пяти дней. Семинар, который длится дольше недели, должен быть разбит на отдельные семинары. Длительные семинары отвлекают людей от работы и требуют более тщательной проработки методики. Семинаров по полдня обычно недостаточно для детального обсуждения, но их можно использовать для подготовки к последующим детальным семинарам.

Если семинар рассчитан на три дня, то полдня можно отвести на обзорное заседание, а остальное время — на детальное обсуждение вопросов. Если обзорное заседание покажет, что для детальной проработки всех проблем недостаточно времени, то можно наметить дополнительные семинары для детального обсуждения.

Руководитель семинара должен следить за регламентом. Повестку дня семинара следует распространить среди участников заранее. Но не нужно ее слишком детализировать. Проще оценить время, необходимое для всего семинара, а не для каждого совещания. Часто дополнительное время, израсходованное на одну проблему, приведет к сокращению обсуждения других вопросов. Если повестка дня сформулирована слишком детально, например в следующем виде:

9:30 Формулировка миссии
10:00 Задачи
11:00 Бизнес-процесс
12.30 ... и т.д.

то при отклонении от нее участники семинара будут чувствовать себя неудобно и могут задаться вопросом, все ли в порядке.

Выделите перерыв на ленч и внесите его в план. Это проявление уважения к участникам. Они смогут рассчитывать на перерыв с 12:30 до 13:30, чтобы позвонить по телефону или сделать другие дела. Перерыв на кофе или чай не нужно планировать на точное время. Его можно сделать по завершении некоторого логического блока обсуждения. Помните также о необходимых регулярных перерывах, особенно если среди участников семинара имеются курильщики.

Чтобы облегчить свою задачу, помощник руководителя может в начале семинара объявить права и обязанности всех участников. Если обсуждение примет несколько эмоциональный поворот, формальные правила помогут разрядить обстановку. В частности, можно жестко регламентировать время докладов и обсуждения. Можно воспользоваться следующими рекомендациями.

- Не допускать параллельных обсуждений во время доклада
- На доклад отводится десять минут; помощник руководителя должен информировать нарушителей, если время закончилось
- Правила хорошего тона диктуют, что недопустимо прерывать других
- Необходимо избегать перехода на личности
- Субординация остается за пределами семинара; участники должны работать вместе
- Мобильные телефоны выключаются. Присланные сообщения можно прочитать во время перерывов
- Каждый получает возможность выступить по любому обсуждаемому пункту. На большом семинаре невозможно услышать мнение каждого участника по каждому вопросу. В таком случае нужно предлагать людям поднимать руки или подавать некоторый сигнал о том, что они хотят высказаться. Помощник руководителя должен следить, чтобы каждый мог высказать свое мнение. Иначе будут преобладать одни и те же личности, в то время как более молчаливые будут редко получать возможность высказаться

На семинаре желательно придерживаться правила "десяти минут", т.е. ровно столько должно длиться обсуждение каждого вопроса. Это означает, что любая тема, которая требует слишком длительного обсуждения, будет помещаться в список открытых проблем для последующего обсуждения в узком кругу. Десятиминутное правило можно использовать, чтобы разрядить конфликты в группе. Если согласие по некоторому вопросу не достигается в течение десяти минут, то обсуждение следует отложить. Спорный вопрос должен быть помечен для последующего решения. Он требует отдельного внимания со стороны руководителя семинара или его помощника. Рассмотрев все аспекты каждого вопроса, можно выработать общую точку зрения. Что делать, если одна тема переходит в другую? Важно сохранить всеобщее понимание проблемы.

Права участников семинара необходимо четко установить, а затем последовательно и тщательно выдерживать.

Ловите момент

При обычном интервьюировании очень сложно удержать участников семинара в пределах темы обсуждения. Может быть, полезно написать текущий вопрос на доске, чтобы напоминать, когда кто-нибудь уклонится от намеченного пути. Это позволит вам оборвать нежелательные ветви обсуждения без отрицательного отношения к лицу, поднявшему этот вопрос. К интересной точке зрения можно будет вернуться позже.

Обратите внимание людей на *время*. Полезно в помещении иметь часы. Это поможет вовремя начинать встречи после перерыва. Если вы объявляете перерыв на 10 минут, возникает соблазн увеличить его до получаса. Тогда из-за каждого перерыва будет потеряно, по крайней мере, несколько минут.

Если вам кажется, что снижается концентрация внимания людей, не бойтесь сделать короткий перерыв (на пять—десять минут) в незапланированное время.

Окончание семинара

После нескольких дней напряженной работы наступает время возвращения каждого участника к своей обычной работе. Самым важным в методе SOMA является достижение общего понимания вопроса между группами разработчиков и пользователей. В этом вопросе ключевую роль играет построение объектных моделей, но немаловажно и то, как происходит закрытие семинара.

Перед окончанием семинара распространяется результирующий документ, в котором все участники ставят свои подписи. Они могут подписать его в случае согласия или не подписывать, если они **принципиально** не согласны с заключительным протоколом. Никаких промежуточных вариантов быть не может. Принципиальные расхождения нужно объяснить, так как это равнозначно задержке проекта. Напоминаем, что протокол семинара является документом, фиксирующим результаты, с которыми согласны все участники.

В случае разногласий предлагается проведение открытых совещаний по проблемным вопросам, чтобы можно было подписать документ с поправками.

Важно, чтобы каждый участник семинара остался удовлетворен. Ни у кого не должно остаться ощущения, что время потрачено впустую. В завершение семинара подводятся итоги и обсуждаются возможные следующие этапы работы.

8.13.9. Использование опросов в контексте семинаров

Многие методы, используемые при обычном опросе, могут быть легко распространены для применения на семинарах. В этом подразделе обсуждается только несколько методов, которые кажутся автору особенно полезными.

Обычно опросы делятся на структурные и сфокусированные. Структурный опрос обычно применяется на начальном этапе разработки и проводится на высоком уровне абстракции. Он охватывает общие темы — широкие, но поверхностные. В результате такого опроса выявляются ключевые объекты и понятия предметной области. Детали не рассматриваются. Такие опросы соответствуют обзорным семинарам (совещаниям).

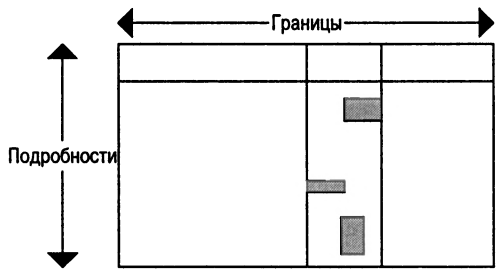


Рис. 8.27. Различные подходы к интервьюированию и прототипированию

Сфокусированные опросы (или детальные семинары) предполагают погружение в некоторую проблему, обычно узкую и глубокую. Это существенно для поиска повторно используемых элементов (серые прямоугольники на рис. 8.27). Аналитики должны выбрать область, в которой сосредоточено 80% полезных функций системы или сложных проблем, и проанализировать ее. Обычно эта область составляет приблизительно 20% от всей системы. Такой же подход обычно применяется при создании прототипов.

Структурные опросы выполняются по заранее подготовленному плану. В начале работы участники должны согласовать повестку дня. Затем по каждой теме повестки дня руководители семинара задают вопросы, производя **зондирование**, делают обзор общих результатов и переходят к следующей теме. В заключение нужно сделать обзор общих результатов и сравнить их с планом, оценивая, достигнуты ли цели. Если цели не достигнуты, осуществляется повторный опрос. Необходимо, чтобы вопросы были открытыми, т.е. не должны допускаться ответы типа "Да" или "Нет", которые закрывают дальнейшее обсуждение или детальную разработку. Зонды — особенно полезные вопросы открытого типа. В табл. 8.4 приводится несколько типов зондов с примерами.

Сфокусированные интервью несколько сложнее. Их форма больше зависит от обсуждаемой тематики. Хороший руководитель семинара планирует и готовит обсуждения заранее. Однако нужно быть готовым к тому, что в процессе семинара план может измениться, т.е. придется адаптироваться к новым условиям. Некоторые знания о предметной области нужны заранее, чтобы облегчить открытое обсуждение, особенно при моделировании бизнес-процессов.

Таблица 8.4. Типы зондов

Тип зонда	Пример
Определительный	Что такое ...?
Дополнительный	Расшифруйте ...
Отражающий	Вы имеете в виду ...?
Изменения режима	Как ваши коллеги смотрели бы на это?
	Вы можете привести более конкретный пример?
Директивный	Почему это так?
	Как?
	Вы могли бы сформулировать мысль более определенно?

8.14. Резюме

В этой главе представлены некоторые детали процесса разработки требований в рамках метода SOMA на основе семинаров. Здесь также сделан обзор других подходов к разработке требований, в том числе методов, основанных на анализе прецедентов. Рассмотрена теория моделирования бизнес-процессов и вопросы их реорганизации.

Для очень больших бизнес-моделей предлагается сформулировать миссию, выделить бизнес-цели и определить их приоритеты.

Рассмотрено несколько проблем, связанных с методом прецедентов, а также введены понятия задач и их сценариев. Прецеденты являются либо отправной точкой для определения требований, либо элементами для повторного использования.

Показано, как метод SOMA поддерживает трассировку проекта. Наконец, описано, как организовывать семинары по инженерии требований.

Изложенная методология проверялась на практике с 1993 года, что подтверждает ее эффективность. Она широко применима в различных методах разработки, в том числе и при использовании UML. Автор рекомендует ее практикам для дальнейшей оценки.

8.15. Дополнительная литература

Представленная в этой главе теория моделирования бизнес-процессов в значительной степени оригинальна, хотя полученные результаты строятся на исследованиях многих других авторов, в частности Якобсона (Jacobson), Флореса (Flores) и Виноградa (Winograd). Часть материала, изложенного в этой главе, сначала была опубликована в журнале *Object Expert*. Многие вопросы изложены также в [329].

Область CSCW рассматривается в работах [342, 793]. В [227] изложено современное состояние проблемы. Работа [442] представляет собой руководство по построению систем автоматизации делопроизводства на основе сочетания реляционных и объектно-ориентированных баз данных, экспертных систем, CSCW, графического пользовательского интерфейса,

мультимедийных систем хранения информации, сетевых технологий и контекстного поиска информации. В книге описана система, действительно обеспечивающая автоматизацию делопроизводства. Однако эта книга является полезным обзором основных технологий, решающих широкий диапазон проблем.

Хотя идею ассоциаций между задачами всегда связывают с методом SOMA, теорию наборов ассоциаций задач разработал Питер Джонс (Peter Jones). Он же показал их эквивалентность с диаграммами последовательности, а также развил многие идеи временного и событийного моделирования, упомянутые в разделе 8.12.1.

В работе [524] очень хорошо описаны диаграммы видов деятельности и принципы их использования. Этот материал изложен лучше, чем в других основных работах по UML, таких как [675], или на Web-узле группы OMG. В работе [715] описаны некоторые ограничения объектного моделирования с помощью UML.

В [480] рассматривается область инженерии требований с акцентом на трассируемость с использованием средства *Requisite Pro,* прецедентов и процесса RUP.

Краткий обзор вопросов инженерии требований приводится в [294]. В работе [499] описано несколько подходов к управлению сеансами.

8.16. Упражнения

1. Почему прецеденты полезны для создания спецификации системы? Почему они не подходят для анализа требований?
2. Дайте определение следующих терминов: агент, исполнитель, диалог, сообщение, действие, прецедент, сценарий.
3. Назовите различия между любыми двумя подходами к инженерии требований.
4. Постройте диаграммы последовательностей и видов деятельности для сценария визита в супермаркет.
5. Какова основная функция руководителя семинара? Что такое семинар?
6. Назовите десять основных задач семинара.
7. Расскажите об использовании голосования на семинаре.
8. Назовите различия между сценариями задач, действиями и прецедентами.
9. Что такое побочный сценарий, компонент сценария и подсценарий?
10. Какова грамматическая структура атомарной задачи? Определите понятие атомарности. Как еще можно определить сценарий?
11. Опишите процесс проверки правильности модели требований:
    а) как проводится тестирование модели бизнес-объектов?
    б) как тестируются задачи?
    в) как тестируется объектная модель задачи?
12. Как преобразовать объектную модель задачи в модель бизнес-объектов? Как модель бизнес-объектов преобразуется в объектную модель реализации?
13. Обсудите использование диаграмм видов деятельности для моделирования бизнес-процессов. Каковы рамки их успешного использования?

14. Что такое карточки классов?
15. Какие методы могут использоваться, чтобы ограничить или расширить обсуждение на семинаре?
16. Что такое зонд? Приведите два примера и назовите тип зонда.
17. **Мини-проект**

Торговый агент нашел партнера для сделки. Он обязан совершить сделку в рамках заданных ограничений. Сначала условия сделки оговариваются устно. Если эти условия удовлетворяют обе стороны, торговый агент оформляет сделку с партнером, согласовывая сроки и стоимость. Соответствующий документ отправляется партнеру.
Информация о сделке передается отделу связей, чтобы его сотрудники могли обеспечить оплату. Этот отдел также обеспечивает руководство процессом и сообщает бухгалтерии о сделке. Счета отправляются по почте. В них указывается дата начала сделки и срок ее действия.
Позиции партнеров и возможные ограничения должны быть урегулированы.

Постройте для этой ситуации модель бизнес-процесса, включающую агентов, диалоги и связанные с ними задачи. Напишите сценарий для одной из задач. Создайте эскиз модели бизнес-объектов, используя соответствующие элементы. Постарайтесь выделить ключевые объекты и четыре типа структурных связей между ними: классификация, использование, композиция и ассоциация. Разработайте несколько карточек классов. Представьте, что вы в составе группы апробации и отладки модели. Постройте диаграммы последовательностей и/или диаграммы видов деятельности.

9

Управление процессом и проектом

Уверяю вас, единственное, что может обеспечить порядок и дисциплину в современном мире, — стандартизированный работник со взаимозаменяемыми частями. Это полностью разрешило бы проблему управления.

Жан Жироду. *Сумасшедшая из Шайо*

Люди часто думают, что процесс разработки программного обеспечения реализуется для автоматизации и ее упрощения. К сожалению, это способствовало развитию сверхбюрократических методов работы, приводящих в лучшем случае к накоплению огромного количества документации, к которой никогда не обращаются ни разработчики, ни их клиенты. Эта макулатура обычно нужна для того, чтобы как-то успокоить чрезмерно нервничающих руководителей, мало понимающих в самом процессе разработки. Часто звучит небезосновательное утверждение, что хорошему разработчику нет дела до процесса как такового. С другой стороны, понятно, что в проектах обычно участвуют не только высококлассные разработчики (и такие же высококлассные руководители). А если процесс надо совершенствовать, его необходимо описать.

Именно поэтому я придерживаюсь золотой середины. Я считаю, что весьма желателен четко определенный, повторяющийся, соразмеренный процесс, но без лишнего размаха и бюрократии. Он не должен умалять роль людей, участвующих в разработке, и сводить на нет важность их творчески созидательной деятельности.

В этой главе исследуется современная система взглядов на объектно-ориентированные процессы разработки, управление проектами и их нормирование. Особое внимание мы уделим быстрой разработке приложений, а также дадим некоторые рекомендации относительно минимального процесса. Данный процесс целиком соответствует модели объектно-ориентированного процесса и представляет собой реализацию процесса OPEN [334, 375], совместимого с процессами DSDM, RUP и XP. Кроме того, он дополняет процесс RUP (Rational Unified Process), предлагая более четкие директивы управления процессом. В нем широко используются прецеденты, однако он является бизнес-процессом, т.е. основан не на прецедентах, а на *контракте или соглашении* (contract driven).

9.1. Зачем придерживаться процесса

Исследования, проводимые группой Standish каждые два года (начиная с 1995), показали, что почти две трети американских проектов по разработке программного обеспечения не доводятся до конца: либо их отменяют, либо они не укладываются в бюджет, либо до производства разработанных программ дело не доходит. Несложно экстраполировать эти (честно говоря, позорные) результаты на остальные части света. Трудно поверить, что наша промышленность и задействованные в ней люди могут терпимо к этому относиться. Разработчикам должно быть стыдно даже просить, чтобы им доверили еще какую-то работу. Исследователи группы Standish также поставили перед собой задачу определить, почему эти проекты, по мнению их участников, так часто заканчивались провалом. И вот какие причины (приведены в порядке важности) были названы:

- недостаточное вовлечение пользователя;
- нечеткие требования;
- бесхозность проекта;
- отсутствие четкого представления о проекте и конкретных задач;
- недостаточное планирование.

С переходом к объектным технологиям некоторые аспекты остаются неизменными: качественная разработка программного обеспечения и правила управления проектом попрежнему играют решающую роль; глубокое знание дела все так же важно, как и мастерство разработчиков; и все так же необходимо понимание действующих правил. Инфраструктура, потребность в документации, порядок проведения тестирования и качественной настройки — все это остается прежним. И это хорошо, потому что таким образом разработчики и руководители могут основываться на том, что уже знают. Тем не менее новый подход все-таки привносит кое-какие изменения. Теперь руководители проектов должны понимать важность инкрементной разработки компонентов наряду с использованием новых языков, сред программирования и компонентов (Java, C++, Beans, San Francisco, CORBA, EJB, COM и т.д.). Как уже неоднократно подчеркивалось, методы объектно-ориентированного проектирования не отделяют процесс от информации — и в этом также состоит их отличие. Вспомните, что более 16% наших затрат возникает из-за изменений в структуре данных. Новый подход, по крайней мере, должен помочь разобраться хотя бы с этими затратами. Ну и, наконец, объектно-ориентированные методы делают возможным гораздо более цельное моделирование и отладку.

Обычные, традиционные процессы часто слишком бюрократичны и склонны к накоплению лишней макулатуры. Эволюционной, итеративной культуре нужны более "легкие" процессы. В нашей консультативной практике в TriReme мы стараемся документально зафиксировать весь процесс разработки для любого клиента на менее чем 100 страницах. Определенно, компаниям, находящимся на пути к объектным технологиям, нужен лучший, более "легкий", но точный и полноценный процесс. Полный охват требований по-прежнему остается первостепенной задачей, поэтому необходим новый подход к самим требованиям, что и предлагалось в предыдущей главе. Необходимы также новые методы объектно-ориентированного анализа. Кроме того, больше внимания должно уделяться планированию и проектированию. Такие организации должны работать под следующими лозунгами.

- Создавайте только те документы, которые действительно будут использоваться (а не просто для начальства).
- Будьте кратки (нормативный *диктат* по модулю — не более 100 страниц).
- Убедитесь, что вы последовательны.
- С самого начала четко уясните требования, но допускайте возможность их изменения.
- Чем быстрее система, тем лучше!

Четко определенный процесс разработки программного обеспечения необходим потому, что организациям мало разрозненных островков успеха, они хотят, чтобы успех повторялся. При четкой организации процесса возникает возможность его усовершенствования и создается основа для нормирования, расчета и оценки, что, в свою очередь, оставляет надежду на улучшение качества продукта. Другими словами, организации хотят усовершенствовать процесс работы. Это вовсе не означает, что нужно придерживаться хваленой модели усовершенствования возможностей CMM (Capability Maturity Model), разработанной институтом инженерии программного обеспечения и изображенной на рис. 9.1 [400], но в принципе такие модели бесспорно представляют чудесную идею. В [425] уровни модели CMM названы весьма условными и произвольными; и действительно, довольно сложно уследить повторяемость чего-то неопределенного: как можно узнать, что же именно повторяется?

9.2. Каковы задачи объектно-ориентированного метода

Чтобы программа имела успех, она должна отвечать требованиям трех категорий заинтересованных лиц: спонсоров проекта, пользователей программы и персонала по ее поддержке. Маловероятно, что требования этих трех групп в точности совпадут. В частности, если программу нужно превратить из концепции и плана в мощное коммерческое приложение, полезное для заказчика, то придется полностью удовлетворить требования первой группы — спонсоров проекта. Когда программа будет готова, эти люди будут судить о ней, основываясь на отзывах, полученных от второй группы, т.е. от пользователей. А отзывы пользователей о ней будут более хвалебными, если она проста в обращении, выполняет нужные в данном деле функции и если поддержка, которую они получают, своевременна и эффективна. Тем не менее до выпуска программы у спонсоров проекта недостаточно информации для оценки

будущего успеха. Ведь они располагают только сведениями о завершенных командой разработчиков проектах и аргументами в пользу программы, изложенными в предварительной документации.

Организации, стремящиеся к объектным технологиям, должны привлекать лучших разработчиков, держаться за них и создавать высококачественные, практичные и удобные в использовании, а также четкие и ясные программы и документацию. Их методы должны отражать существующий порядок (т.е. то, что сейчас успешно выполняется), а не насаждать бюрократические порядки, необходимость которых поймут немногие разработчики. Такие методы полезны лишь для неопытных разработчиков.

Компьютеры обеспечивают решение многих проблем, возникающих в промышленности и торговле. Теоретически, новая разработка должна представлять собой новое решение неразрешенной проблемы. А на практике решение может уже существовать, просто его должным образом не донесли до заинтересованных лиц. В любом случае в анализе и разработке новой программы будет иметься неопределенность. Но эффективный анализ и проектирование быстро уменьшат эту неопределенность настолько, что с ней будет легко справиться. Чтобы успешно с ней разобраться, следует воспользоваться уже существующими и принятыми технологиями и программами. А именно, в качестве основы разработки как на начальном этапе, так и на протяжении всей работы над проектом следует использовать стабильные платформы. По этому поводу Буч [99] приводит цитату:

> Любая работающая сложная программа является результатом развития более простых работающих программ... Сложные программы, написанные с нуля, никогда не работают и не могут быть доведены до рабочего состояния. Вам придется начать сначала, взяв за основу простую работающую программу.

Рис. 9.1. Модель CMM института инженерии программного обеспечения

Необходимо разрабатывать эволюционирующие программы по принципу поступательного развития, а не предлагать решения в духе "большого взрыва". Предлагаемый подход не только следует нормам, на которые указывал Буч, но и позволяет пользователям и спонсорам комментировать промежуточный продукт и вносить больший вклад в проектирование разрабатываемой программы.

Спонсоры должны быть осведомлены о методе разрешения неопределенности и об эволюционном подходе к созданию программ. Такие цели нужно ставить в планах проектов и отчетах по результатам системного анализа. Кроме того, следует заранее оговорить, что должно быть представлено по завершении каждого цикла разработки. Ожидаемые результаты необходимо скоординировать с активным участием пользователей и спонсоров в процессе формулировки требований, проектирования и разработки программ. Так как это стало обычной практикой в рамках некоторых успешных существующих программ, всем организациям следует принять ее как осознанную политику.

Трудность проведения такой политики заключается в том, что степень участия пользователя или спонсора меняется в зависимости от разрабатываемого продукта. Многие разработчики графического пользовательского интерфейса GUI с радостью вовлекают в работу пользователей и спонсоров с их комментариями. К тому же инструментальные средства разработки GUI позволяют быстро внести косметические изменения в ответ на пользовательские запросы. Тем не менее такого же пользовательского энтузиазма труднее добиться в области обработки групповых операций и расчетов без графического представления процесса.

И все же следует официально способствовать более активному участию пользователя на протяжении всего процесса развития проекта. Более активное вовлечение пользователя повышает степень общественного интереса к организации-разработчику, привлекает спонсоров и новых пользователей, а также обеспечивает более открытую и эффективную связь с упомянутыми кругами. Это позволит пользователям четче сформулировать свои требования, а разработчикам — точнее понять их. В результате организация-разработчик будет выпускать программы, действительно *необходимые* в торгово-промышленной деятельности, а не желаемые или навязываемые.

Хороший метод разработки программы должен поддерживать дисциплину разработчиков на разумном уровне, чтобы их производительность не страдала от ненужной бюрократии и бесконечных компонентов, не вносящих вклад в конечный продукт. Недопустимо также считать, что в методе заключены решения всех проблем, потому что бывают непредвиденные ситуации и типы задач. Метод должен способствовать достижению производственной цели — постоянному усовершенствованию процесса. Это значит, что некоторые средства оценки качества продукта и эффективности процесса должны быть утверждены как обязательные, чтобы можно было определить, продвинулись ли разработчики вперед и насколько. Производительность повышается тогда, когда требования метода отвечают шагам, которые идеальный разработчик предпринял бы в реальном мире под давлением обстоятельств, не пользуясь преимуществами метода. Компоненты должны быть естественны, существенны и должны действительно использоваться кем-то для какой-то полезной цели. Но многие методы требуют создания компонентов, которые могут быть использованы только в определенных обстоятельствах или вообще нужны лишь для того, чтобы оградить руководителей от возможной критики. И методы, принимаемые правительствами, часто являются наиболее злостными нарушителями в этом плане, потому что слишком часто *видимость* того, что работа выполняется как следует, важнее, чем надлежащее ее выполнение, и для чиновников, и для политиков.

Метод должен способствовать удовлетворению реальных промышленных потребностей, помогая проследить, чтобы качество поддерживалось и улучшалось, требования были уяснены точно и полностью, потребности не подменялись желаниями, а также проследить за соблюдением сроков и производительностью разработчиков. Данный метод должен поощрять создание гибких систем, которые можно изменять по ходу дела с минимальными затратами. Кроме того, с его помощью проще контролировать и сокращать время до поступления продукта на рынок, чтобы пользователям не пришлось годами ждать определенной программы, в то время как требования неумолимо продолжают изменяться.

Метод, предложенный в этой главе, сокращает время до поступления продукта на рынок путем применения эволюционного подхода к разработке, основанного на временных рамках, когда разработка важных уровней программы должна укладываться максимум в шестимесячный срок. Менее значительные модули можно создать и добавить позже. Следует признать, что при таком подходе не стоит ожидать, что вся огромная, сложная система выстроится в мгновение ока и что некоторые программы не развалятся на составные части. Мне, например, не хотелось бы лететь на самолете с бортовыми системами, разработанными лишь на 80%. Тем не менее, судя по моим наблюдениям, большинство коммерческих программ подчиняется этому подходу. На самом деле, однажды старший программист-разработчик компании "Боинг" сказал мне, что авиалайнеры-гиганты действительно строились с использованием этих самых эволюционных методов. К счастью, промежуточные продукты у них испытываются на стенде. Они их так сразу не запускают!

Объектно-ориентированный метод должен повышать производительность, непосредственно поддерживая и поощряя повторное использование программ в новых разработках и применение подходящих мощных инструментальных средств. Безусловно, прибыль от конечного продукта должна компенсировать стоимость разработки и инструментальных средств. Если же взглянуть на это с точки зрения Буча [98, 99], согласно которой объектно-ориентированная разработка обоснована, в основном, тем, что позволяет нам лучше справиться со сложностью систем, все сводится к более быстрому созданию более ценных, больших и сложных программ. Их качество также не должно пострадать, и это означает, что сам продукт, а также показатели его разработки и финансовые показатели должны быть объединены в единое целое. Даже от наилучшего метода не будет толку, если его требования покажутся разработчикам обременительными; они обязательно найдут способ обойти их. Хороший объектно-ориентированный метод разработки должен быть прост и удобен в использовании — и применяем.

Объектно-ориентированный процесс должен работать незаметно для пользователя, применяя объектно-ориентированное моделирование от начала до конца. Это само по себе улучшит отладку, хотя техника соединения модулей должна быть подробно разработана, о чем уже говорилось в главе 8.

Разумный объектно-ориентированный процесс сократит время до поступления продукта на рынок, при условии что управление проектом и порядок разработки компонентов согласованы, а повторное использование программ организовано как следует.

Завершенный объектно-ориентированный процесс должен включать следующее.

- моделирование требований и производственного процесса

  — семинары, ролевые игры и проверку корректности

- легкую, настраиваемую среду разработки процесса

- управление проектом
  - — модель предстоящих работ, средства, показатели, настройка, тестирование и т.д.
  - — организация повторного использования программ
- компонентную архитектуру
- системную спецификацию (техническое описание системы)
  - — прецеденты/операции, шаблоны процесса, UML и т.д.
- компонентную разработку и декомпозицию
- тестирование на протяжении всего времени работы над программой
- организацию контроля качества, ввода в эксплуатацию и настройку
- шаблоны процессов

Термин *шаблон процесса* (process pattern) относится к любому процессу разработки и сопровождения программного обеспечения. Такие шаблоны часто встречаются, их легко узнать и им охотно дают названия. Шаблоны процесса имеют такую форму.

```
Если вашей целью является A,
а вы находитесь в положении B,
попробуйте сделать C
  (но имейте в виду предпосылку P, риск R,
   побочный эффект S, масштабы времени T и т.д.)
```

Главное — поглощать мудрость стратегии и тактики создания программного обеспечения перевариваемыми порциями. Одна из проблем многих методик разработки программ заключается в том, что в них предполагается определенное исходное положение: допустим, что разработчик проектирует продукт с нуля. На практике же такое бывает редко. Предлагаемый процесс разработки должен принимать во внимание различные исходные состояния. Мы стараемся выразить сущность нашего опыта работы с процессами в отдельных шаблонах. Каждый из них имеет приведенную выше форму "производственного правила": триггер⇒ действие. Эти действия часто ставят подзадачи, для которых находятся другие шаблоны, так что шаблоны можно объединить в общую цепочку и получить шаблоны более высокого уровня.

При выборе шаблона процесса найдите все шаблоны, применимые к вашим задачам и конкретному проекту. Они послужат основой вашего плана. Может быть несколько шаблонов, описывающих как разные аспекты плана (например, создание команды или оценивание временны*х* рамок), так и основные аспекты всего плана в целом. Поскольку они сформулированы не строго, вам придется сообразить, как их скомпоновать и объединить. Это не механический процесс. Выбирая шаблон (или, возможно, объединяя несколько), вы приступаете к проектированию обычного процесса. Каждый шаблон ставит свои подзадачи, для решения которых можно подобрать другие шаблоны. Выбирая и применяя их, вы детализируете свой план. Шаблоны процесса лучше всего работают в структуре общего процесса.

Описанный в этой главе процесс уходит корнями к процессам MOSES, SOMA, DSDM и Catalysis. Это полностью объектно-ориентированный процесс:

- предназначен для реализации проектов, моделирующих мир с помощью объектов;
- любой новый код будет объектно-ориентированным (за исключением, возможно, оболочки);
- сам процесс описывается как набор взаимодействующих объектов;
- для него используются объектно-ориентированные метрики;
- это эволюционный и итеративный метод.

Для достижения максимальной гибкости он содержит и высокоуровневые метапроцессы, описанные в этой главе, и представленные в виде компонентов шаблоны отдельных процессов, в том числе соответствующие субпроцессам обработки требований и проектирования, описанным в предыдущих главах.

С точки зрения быстрой разработки приложений этот процесс можно назвать объектно-ориентированным вариантом DSDM, принципов которого он строго придерживается. С тем же успехом, с точки зрения разработчиков, его можно рассматривать как дополнение к технологии RUP (Rational Unified Process); хотя мы предлагаем бо́льшую гибкость и управление процессом на протяжении всего времени его реализации.

В отличие от технологий DSDM и RUP, этот процесс более доступен. Он более легок и при этом сохраняет точность. Перефразируя Эйнштейна, можно сказать, что он "настолько прост, насколько это возможно, но не более". В нем содержатся многие идеи экстремального программирования XP. Например, мы поощряем использование такой техники, как парное программирование и использование коротких и быстрых циклов.

9.3. Классические модели жизненного цикла

В этом разделе предлагается критический обзор моделей жизненного цикла, используемых в инженерии программного обеспечения до появления объектных технологий.

9.3.1. Каскадная модель, V- и X-модели

Существует несколько способов обеспечения аналогии разработки проектов в области информационных технологий с другими промышленными или природными процессами. Самая старая и наиболее известная модель — это модель "водопада", или каскадная модель, которую часто называют эквивалентом модели процесса в строительной промышленности. Наиболее раннее документальное подтверждение этой идеи вы найдете в [671]. Суть ее в том, что приступать к анализу можно только после полного определения всех требований. По завершении анализа — и только тогда — можно начинать логическое проектирование и т.д. Итерации иногда допускаются, но лучше их избегать, поскольку это дорого обходится, и чем большее количество фаз жизненного цикла будет пройдено при возвращении назад, тем выше будут затраты (рис. 9.2).

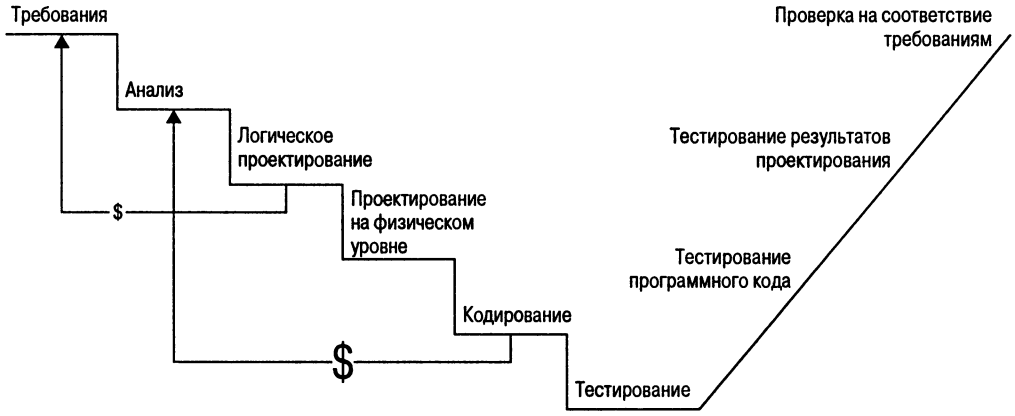


Рис. 9.2. Каскадная модель с несколькими итерациями и ее расширение до V-модели. Чем большее число этапов придется переработать, тем выше будут затраты. Тестирование проводится слишком поздно

С каскадными моделями тесно связаны так называемые V-модели. Они соотносят подэтапы проверки с предшествующими этапами "водопада". В процессе выполнения модульного тестирования проверяется код; интеграционное тестирование и тестирование в предельных режимах позволяют проверить проектное решение; и, наконец, опытные испытания и анализ функционирования внедренной системы дают возможность проверить соответствие системы модели анализа и техническим требованиями. Чем позже в процессе тестирования будет обнаружена ошибка, тем дороже обойдется ее исправление.

Известно, что разработчики программного обеспечения никогда не поддерживали каскадную модель создания программ. Ее придумали организаторы проектов, чтобы контролировать график выполнения работ и оплаты. Как только фирма-исполнитель подготовила отчет по результатам анализа, работу можно оплатить. Естественно, покупатели часто считали, что раз уж они заплатили за что-то, оно непременно должно работать, поэтому от проведения повторного анализа воздерживались, если только организатор проекта не проводил его бесплатно (и, конечно же, весьма неохотно). В результате фирмы-исполнители разработали жесткие методы, чтобы сделать произведенный на каждом этапе компонент максимально полным, вплоть до прочно утвердившейся практики невыполнения требований, которые клиент четко не сформулировал (даже если их необходимость очевидна). Эти методы состояли в основном из эвристических правил и технологических инструкций или контрольных таблиц. Нынешнее весьма напряженное положение говорит об очевидной необходимости более гибких подходов, допускающих создание прототипов и быструю разработку при помощи методов с большим числом итераций.

В [390] предложено расширить стандартную V-модель до так называемой X-модели, улучшив ее за счет принципа повторного использования. Она изображена на рис. 9.3 с некоторыми терминологическими изменениями, внесенными для большей ясности. Эта модель отслеживает передачу компонентов и структур в хранилище, используя аналогию между обработкой данных и ведением учета. Она, безусловно, более совершенна, чем традиционная V-модель, и действительно проясняет вопрос повторного использования. И все же я считаю, что такие модели не подходят для современной объектно-ориентированной разработки, потому что не обеспечивают тестирование на ранних стадиях, создание прототипов и итеративную разработку.

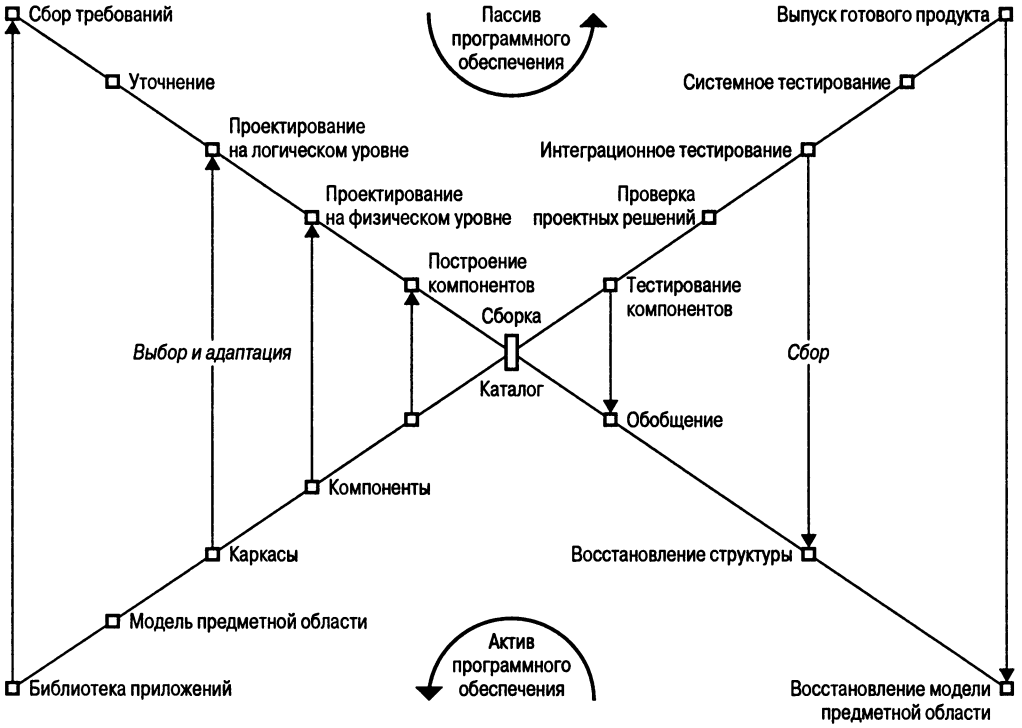


Рис. 9.3. X-модель

9.3.2. СПИРАЛЬНЫЕ МОДЕЛИ

Барри Боэму (Barry Boehm) из компании TRW приписывается первая формализация модели итеративного процесса. Основная его идея заключалась в том, что разработка программного обеспечения должна определяться риском и конечным продуктом. Его спиральная модель представлена на рис. 9.4 в модернизированном виде. А объясняется она так. Разработка начинается с обзора и исследования имеющегося положения в данной отрасли с должным вниманием к существующим системам. Нужно проследить и оценить ситуацию, чтобы можно было проанализировать риск при внесении последующих изменений. Если позитивный риск значительно превышает негативный (т.е. значительно преобладают обстоятельства, говорящие в пользу продолжения работы), следующим этапом будет планирование работ. Далее реализуется небольшой каскадный процесс разработки, в рамках которого может быть создана полнофункциональная система, ее прототип, пилотный вариант или отдельный компонент окончательной системы. Далее следует пересмотреть ситуацию, чтобы определить необходимость дальнейшей разработки и при ее наличии спланировать следующий шаг. Этот процесс может повторяться неограниченное или предварительно определенное число раз. Несомненно, эта модель более прогрессивна, чем каскадная модель. Многие критики утверждали, что это в действительности не итеративный процесс, а линейный процесс с повторениями [378]. На самом деле его можно рассматривать как нечто, напоминающее рекурсивный "водопад", который существенно отличается акцентом на рисках и производством

используемых продуктов на каждой стадии разработки. В [92] отмечено, что эта модель не подходит для работы с внешними организаторами проектов. Главный ее вклад — акцент на анализе рисков.

Рис. 9.4. Современная версия спиральной модели Боэма

9.3.3. Модель типа "фонтан" и процесс MOSES

Брайан Хендерсон-Селлерс (Brian Henderson-Sellers) и Джулиан Эдвардс (Julian Edwards) создали модель жизненного цикла, построенную по аналогии с фонтаном, а не водопадом (рис. 9.5). В ней неявно выражена спиральная модель. Подчеркивается, что в этой модели фазы разработки частично перекрываются, уровни и подсистемы спонтанно выделяются на разных стадиях разработки, а отдельные компоненты разрабатываются неравномерно. Эта модель позволяет разделить объекты предметной области и программные объекты, что должно способствовать повторному использованию кода. В центре фонтана находится "бассейн" — накопитель объектов для повторного использования, в который попадают готовые компоненты. Есть тут и промежуточные накопители. Новые компоненты проходят через тестирование и использование программы к сопровождению и дальнейшей разработке. И новые, и уже существующие компоненты могут проходить фазу обобщения и повторной оценки, прежде чем снова попасть в накопитель повторного использования.

Существует три варианта реализации фонтанной модели, которые соответствуют следующим ситуациям:

- все классы новы и проходят все этапы жизненного цикла;
- возможно повторное использование, т.е. классы модифицируют, объявляют и создают производные классы, а не разрабатывают и строят новые;
- система попросту компонуется из существующих классов, основываясь на тщательном анализе предметной области.

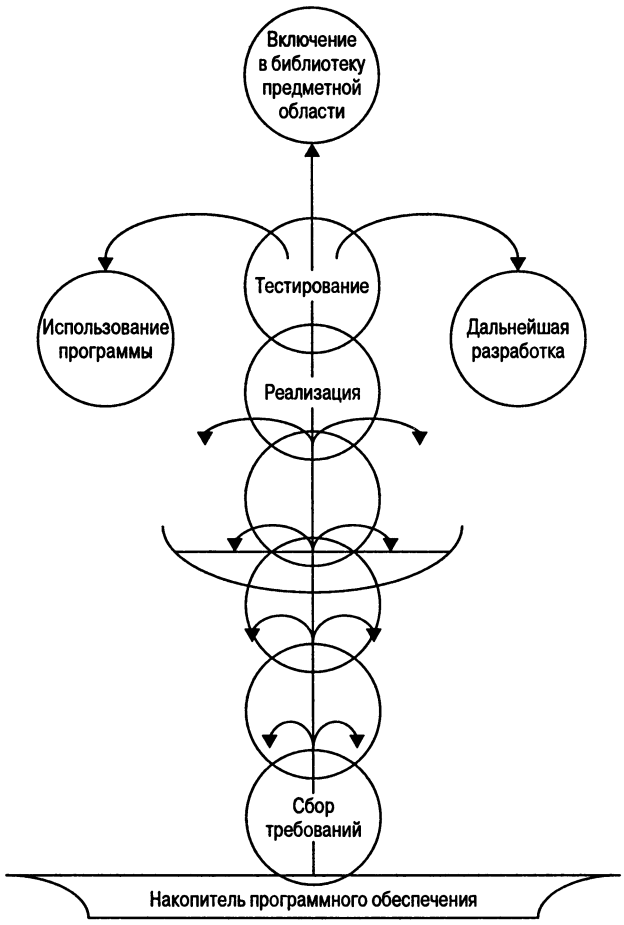


Рис. 9.5. Модель типа "фонтан"

Первый сценарий подходит несведущим в объектно-ориентированных методах организациям, а последний вряд ли можно будет воплотить в обозримом будущем. "Фонтан" — полезная модель, но для самостоятельного проведения процесса разработки она недостаточно детализирована. Поэтому далее мы используем ее идеи в более инструктивном, но гибком подходе.

Процесс MOSES определяет виды деятельности в соответствии с фазами, представленными в виде таблицы, подобной табл. 9.1.

Этап планирования — это повторение стадии составления бизнес-плана для более короткого периода времени. Исследование включает сбор существующих текстовых источников, предшествующих моделей и характеристик существующих классов, которые можно повторно использовать, а также проведение опросов или семинаров и определение возможностей и ограничений. Этот процесс переходит на стадию спецификации, где опять-таки проводится анализ и совместные семинары. Отчет о результатах системного анализа готовится с первой же

итерации. Последующие итерации уже могут учитывать ограниченные возможности компьютеров. На стадии реализации логическая схема объектной модели превращается в проектное решение на физическом уровне, которое обычно ориентировано на конкретный язык и завершается написанием кода. Модель проектирования должна включать решения о выборе базы данных, преобразовании схем классификации в схемы наследования, использовании членов со спецификаторами `public`/`private`/`protected`/`friend` и т.д., а также о реализации подсистем, параллельных связях между объектами и архитектуре. Тут же проводится модульное и интеграционное тестирование. Фаза рецензирования включает дальнейшее тестирование реализации прецедентов и определение качества продукта и процесса.

Таблица 9.1. Упрощенные фазы и виды деятельности процесса MOSES

Фаза	Вид деятельности
Планирование	Оценка, разработка графика, постановка задачи, определение необходимых ресурсов, анализ риска
Исследование	Уяснение требований, сбор информации, понимание проблемы, определение масштабов задачи
Спецификация	Анализ, логическое проектирование на высоком уровне
Реализация	Физическое проектирование на низком уровне, кодирование, тестирование
Рецензирование	Проверка качества, оценка плана проекта, оценка показателей, тестирование по заданным сценариям

Мы не следуем в точности этой модели, потому что процесс исследования происходит не только в заданных временны*х* рамках или временны*х* блоках (в пределах отрезков времени, выделяемых на выполнение определенных объемов работ), но и за их пределами. На семинарах определяются требования, применимые ко всем итерациям в рамках, как минимум, одного временно*г*о блока; но рецензия любого пользователя в этом временно*м* блоке фиксирует дополнительные требования — по крайней мере в общих чертах. Если допустить, что планирование — это главным образом установка цели и определение сроков, то становится ясно, что цель определяется на семинарах и лишь корректируется в ходе созидательных работ. Фаза рецензирования тоже распределяется между временны*ми* блоками (тестирование на основе прецедентов) и стадией оценивания (которой соответствует фаза рецензирования в процессе MOSES). Подход, основанный на идее временны*х* блоков, мы подробно опишем в следующем разделе.

9.3.4. Фрактальные модели, модели типа раковины и игры в пинбол

Другие предложения и варианты "спирали" предназначались для поддержки создания прототипов и/или преобладания методологии быстрой разработки приложений RAD. В [538] описывается фрактальная модель по Форте (Forte), которая основывается на наблюдении, что разработка самоподобна на разных уровнях детализации: класса, уровня, системы и т.д. В [378] такой подход подвержен критике на том основании, что фракталы бесконечно рекурсивны и

детерминированы, а разработка, по мнению авторов, — нет. Тем не менее я нахожу полезной эту аналогию и смею надеяться, что ее будут использовать в будущих методах. Вопрос детерминизма всегда был спорным: вспомните, например, дискуссии о скрытых переменных в квантовой физике или прения по поводу открытых и закрытых систем в кибернетике. Микропроцесс Catalysis для создания спецификаций и разработки компонентов в высшей степени фрактален.

В литературе представлено несколько вариантов спиральной модели — например, "раковина", являющаяся на самом деле другим, более подробным схематическим изображением "спирали" (кстати, первоначальное изображение Боэма имело именно такую форму). Мой собственный опыт "общения" с такими моделями вскоре показал, что, хоть от этого модельного представления и была какая-то польза, реального руководства к действиям оно не давало. Надо было разработать что-нибудь получше. В этой главе вы увидите результаты моих попыток прийти к по-настоящему объектно-ориентированной модели жизненного цикла. На сегодняшний день этот подход уже использован в большом количестве коммерческих проектов и принят с некоторыми изменениями в некоторых крупных организациях.

Существуют модели, основанные на представлении проекта в виде стола для игры в пинбол (настольная игра, в которой игрок, выпустив с помощью поршня шарик, пытается попасть в лунки, расположенные на игольчатой поверхности), где результаты "прыгают" среди видов деятельности, задающих им разные направления и энергию. Эта модель не настолько безумна, как кажется на первый взгляд, но автор опасается появления рок-оперы под названием "Разработка программного обеспечения", в основе которой будет лежать эта идея.

9.4. Семинары, временные блоки и эволюционная разработка

Быстрая разработка приложений, или технология RAD, стала популярным подходом к разработке в конце 1980-х годов, но ее происхождение по-прежнему неясно. Принцип совместной разработки приложений JAD (Joint Application Development), состоящий в проведении регулярных семинаров и использованный в компании IBM, безусловно, является существенным компонентом этой модели. Быстрое создание прототипов использовалось годами и также повлияло на подход RAD. Иногда появление этого названия связывают с именем Джеймса Мартина (James Martin), но первым формальным названием этой модели, кажется, было имя RIPP (Rapid Iterative Production Prototyping), означающее быстрое итеративное создание прототипов. Этот метод позднее рекламировался под броским лозунгом "Ваша программа за 128 дней — или мы возвращаем вам деньги". За этим последовало множество патентованных клонов, и вскоре каждый использовал свою версию с собственным именем. Общим для них всех было использование семинаров и временных блоков. "Временной блок" строго ограничивает итерации, чтобы обеспечить контроль руководства над созданием прототипов. Обязательна небольшая команда участников процесса, а протестированный прототип является одновременно конечным результатом и готовым компонентом. Методы RAD различаются по степени вовлечения пользователей в процесс создания прототипов и по используемым инструментальным средствам. Одни, как в случае RIPP, настаивают на использовании языков четвертого поколения и архивов, а другие делают особое ударение на CASE-средствах и генерировании кода. Автор хочет придать особое значение построению связующего моста между пониманием пользователей и разработчиков, а также объединению их усилий.

Технология RAD очень выгодна, независимо от того, используются ли там объектно-ориентированные инструментальные средства, и количество практических доказательств тому все растет. В [690] исследовано 34 проекта и отмечено значительное увеличение простоты использования и соответствия выставленным требованиям. Кроме того, имеются доказательства повышения удобства сопровождения, но только когда одноразовым прототипам не позволяют развиться в конечный проект. Авторы не нашли подтверждения "общему мнению, что быстрое прототипирование нельзя использовать для разработки больших систем", и посоветовали использовать объектно-ориентированный подход в рамках различных дисциплин процесса, например, описанных в [370] или приведенных в этой главе. При использовании семинаров затраты значительно ниже, чем при проведении многочисленных опросов пользователей; кроме того, экономится много времени. Семинары придают структурированность определению требований и их анализу, и в то же время они динамичны и согласованы. Они задействуют пользователей и часто разрушают границы между организациями. Семинары помогают определить потребности и назначить приоритеты, а также решить спорные вопросы. С самого начала поддерживается идея привлечения пользователей к разработке продукта, и это дает хороший толчок для проведения анализа. Решения и компромиссы записываются на семинарах; первый из этих семинаров — это первая возможность изменить ожидания и позицию пользователей.

Методика временны*х* блоков имеет свои преимущества. Она обеспечивает контроль руководства над волновыми эффектами и спонтанными итерациями. Достигается это за счет установки четких сроков для осуществления прототипирования и за счет использования маленькой команды. При этом и конечным результатом процесса, и его компонентом является пригодная к использованию система. Фазы реализации, развития и сопровождения не разграничиваются, как при традиционных подходах, в которых при утверждении проекта часто не учитываются затраты на сопровождение.

Временны*е* блоки позволяют решить такие проблемы управления.

- Желания против потребностей. Эта проблема решается с помощью обязательной установки приоритетных требований в процессе переговоров между пользователями и разработчиками. Пользователи и команда, работающая над проектом, должны сконцентрироваться на реальных потребностях.
- Изменение функциональности. В традиционном жизненном цикле большой разрыв между составлением спецификации и этапом сдачи работы может привести к просьбам о реализации дополнительных функций. Использование временны*х* рамок ослабляет эту тенденцию.
- Мотивация команды разработчиков. Разработчики видят ощутимый результат своих усилий.
- Привлечение пользователей на каждом этапе смягчает их реакцию на появление конечного продукта.

Этот подход сокращает время, необходимое для выхода продукта на рынок, не по волшебству, которое делает сложные вещи простыми, а за счет предоставления пригодной к применению основной части системы не позднее, чем через несколько месяцев.

Очень важно не потерять доверия, рассчитывать на успех и оправдать ожидания в процессе разработки. Вот какими средствами это достигается. Пользователей нужно предупредить,

что наскоро разработанная модель может скрывать огромные сложности. Время, затрачиваемое на создание рабочей системы, пропорционально сложности выполняемых ею задач. Кроме того, следует баловать пользователей новыми версиями. При моем подходе они соглашаются с выбором приоритетных задач, но разработчики должны показать им, что углы, срезанные, чтобы уложиться в сроки, весьма несущественны. Таким образом разработчики могут позволить себе принять разумные изменения требований, при условии исключения существующих низкоприоритетных требований по взаимной договоренности. Таким образом, оправдание ожиданий — это главная задача руководителя проекта. И обычно, если ею пренебрегают, проекты проваливаются.

Этот метод предотвращает появление тормозящего эффекта от проведенного анализа, ошибок из-за задержек, побочных требований и смягчает реакцию пользователей на появление конечного продукта. Это дает лучшую мотивацию команде разработчиков, чем каскадный подход.

Единственное замечание в адрес эволюционной и итеративной быстрой разработки — ее суммарные выгоды могут во многом вызываться эффектом Хоторна (Hawthorne), поскольку, уделив чуть больше внимания пользователям и разработчикам, можно повысить производительность независимо от используемого метода. Кроме того, возможно, причиной успеха является просто хорошая подготовка и мотивация пользователей и разработчиков, так как методология RAD предполагает привлечение опытных квалифицированных кадров. Не могу опровергнуть ни одного из этих утверждений, но все же я по-прежнему уверен в пользе методологии RAD.

Предлагаемый процесс объединяет технологии JAD, RAD и ОТ. В одной из компаний, где я работал, технология RAD и объектно-ориентированное программирование были введены одновременно, но независимо друг от друга, со значительным улучшением производительности в каждом случае. Семинары обеспечили большинство из упомянутых выше преимуществ, но, к сожалению, они проводились согласно структурным методам и обычно давали усредненные модели объектов и вызывали чисто функциональную декомпозицию. Разработчики обычно говорили: "Классные картинки" — и, подшив их куда-то, шли искать пользователя, который смог бы помочь им создать какой-то прототип. Жизненный цикл на основе контрактов придумали отчасти для того, чтобы помочь этой компании согласовать технологию RAD с объектно-ориентированным программированием.

9.4.1. Принципы динамической разработки систем

Метод динамической разработки систем DSDM (Dynamic System Development Method) — это "контур элементов управления для быстрой разработки приложений" [726]. В отличие от процессов OPEN, RUP или SOMA, здесь не предписывается использование конкретных методов. Метод DSDM имеет элементарную, но высокоуровневую модель процесса, которую можно изменить индивидуально для любой организации в соответствии с ее требованиями. Разработал этот потенциально образцовый подход в 1994 году консорциум 17 британских пользовательских организаций, ныне насчитывающий более тысячи членов во всем мире. Он воздерживается от приверженности к какому-либо стилю разработки, и поэтому версия 2 уже не содержит никаких ссылок на объектно-ориентированные технологии. Наверное, это и к лучшему, потому что те немногие замечания об объектной ориентации, что были сделаны в версии 1, были во многом ошибочны. С другой стороны, с принятой консорциумом DSDM точкой зрения (что объектно-ориентированный подход — вполне обыкновенная методика), на мой взгляд, можно поспорить, потому как я считаю объектную ориентацию общим методом

представления информации. Метод DSDM основан на следующих девяти базовых принципах, которые полностью реализованы в описанном ниже процессе.

1. Активное вовлечение пользователя обязательно.
2. Команды имеют право принимать решения.
3. Важно чаще выпускать новые версии продукта.
4. Польза для дела — важный критерий принятия разработанных компонентов.
5. Итеративное развитие и инкрементная разработка компонентов необходимы, чтобы прийти к точному коммерческому решению.
6. Все изменения в процессе разработки обратимы.
7. Требования определяются на высоком уровне.
8. На протяжении всего жизненного цикла постоянно проводится тестирование.
9. Важны сотрудничество и содействие всех заинтересованных лиц.

Эти принципы позволяют организации определить соответствие отдельного проекта общему подходу. Например, если вы точно знаете, что ни при каких обстоятельствах в вашем проекте не будет участвовать группа пользователей, забудьте о нем. Конечно, часто можно воспользоваться помощью представителей пользователей, а не самих пользователей. Хоть это и не идеальный вариант, обычно это лучше, чем возвращение к старому, негибкому и неэффективному пути.

На рис. 9.6 изображена модель процесса разработки DSDM. Процесс начинается с оценки осуществимости проекта и изучения предметной области, за чем следуют три итеративные и перекрывающиеся фазы. В основе этого процесса лежит преимущественно спиральная модель.

Метод DSDM не совсем подходит для объектно-ориентированной разработки, потому что ему недостает гибкости объектно-ориентированного подхода. Кроме того, он не поддерживает концепции проектирования или разработки на основе контракта или обязанностей. Чтобы сполна воспользоваться преимуществами объектных технологий, придется модифицировать жизненный цикл. Данный метод не предполагает также формального разбиения программы на более мелкие проекты, которое используется, например, в методах OPEN или SOMA. Кроме того, в методе DSDM не различаются должным образом жизненные циклы процесса и продукта. Эти проблемы обсуждаются в разделе 9.6, где мы увидим, что фазы метода DSDM можно привести в соответствие с объектно-ориентированной, определяемой соглашениями моделью жизненного цикла, о чем мы поговорим позже.

Что действительно полезно в методе DSDM — это структура, которую можно наполнить конкретными политическими и организационными характеристиками проекта. Заключение во временные рамки (3-й принцип) означает, что этапам соответствуют разработанные в результате компоненты, что четко ориентирует этот подход на продукт. Тестирование проводится на протяжении всего жизненного цикла, а не в качестве заключительного этапа, что ведет к значительному повышению качества и сокращению количества сюрпризов при реализации. Очень важно, что предоставляемые продукты и документация имеют минимальные объемы и в то же время обеспечивают достойное качество и возможность дальнейшей разработки и сопровождения. К сожалению, без объектно-ориентированного подхода нельзя гарантировать простоту сопровождения системы, несмотря на последующее развитие требований, что уже объяснялось в главе 1.

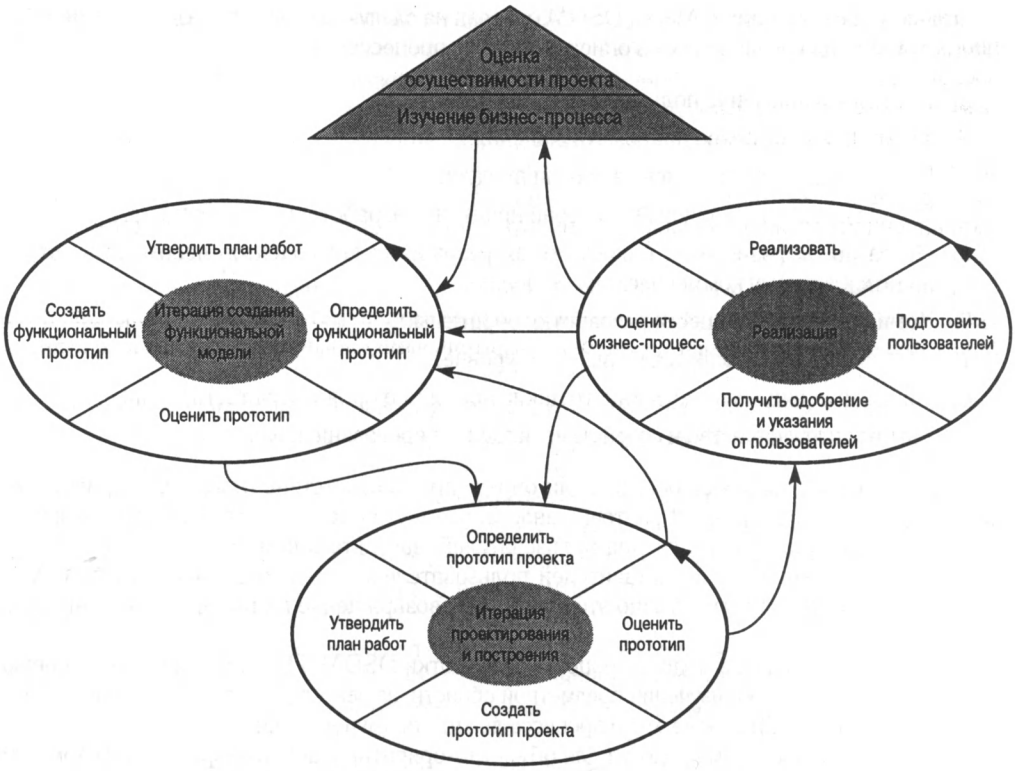


Рис. 9.6. Модель процесса разработки DSDM [726]

Мы уже отмечали, что метод DSDM ориентирован на продукт. И все же, я думаю, нам нужно более строго добиваться производства действующих компонентов, чем при методе DSDM. Последнему приходится идти на компромисс, потому что традиционно разработанные продукты обычно состоят из более крупных компонентов, чем продукты объектно-ориентированной разработки. Так, в методе DSDM конечным продуктом временного блока может быть модель данных. А в объектно-ориентированном методе RAD, как мы убедимся позже, все произведенные объектные модели доводятся до уровня выполняемых модулей (хотя бы потенциально). Но в любом из этих методов ориентация на продукт является более гибкой, чем при проблемно-ориентированном подходе, в рамках которого низкоуровневые задачи распределяются индивидуально между членами команды. В условиях последнего метода команде намного труднее вносить поправки, чтобы уложиться в сроки, и вот тут с наибольшей силой действует второй принцип (права разработчиков).

Девятый принцип не только говорит о необходимости включения в команду и разработчиков, и пользователей, он также требует выполнения определенных организационных принципов. Стэплтон [726] указывает следующее.

> Некоторые организации искусственно создают границы между разными частями отдела информационных технологий. Разработчикам приложения *незачем* слишком быстро собирать систему, если управляющий персонал не стремится к скорейшему ее запуску и откладывает его из-за собственных неувязок в приоритетах (*выделение добавлено*).

Таким образом, если организация поддерживает традиционный подход к тестированию программного обеспечения пользователями UAT (User Acceptance Testing) или ее группа сопровождения подчиняется не тому же руководству, что и разработчики, ей будет труднее и куда менее выгодно внедрять методику RAD.

В методе DSDM, как и в методе OOSE [417], основное внимание уделяется интерфейсу системы. Это идеально подходит для простых информационно-управляющих систем или для спецификации такого машинного оборудования, как торговые автоматы или коммутаторы дистанционной связи, но это далеко не то, что нужно для более сложных систем (см. главу 8). Метод SOMA позволяет при выработке требований на этапе анализа "проникать" в аспекты пользовательского интерфейса, чтобы создать общее понимание того, как же на самом деле будет работать система. Возможно, из-за этого ограничения метод DSDM не признали подходящим для приложений с какими-либо вычислительными сложностями. Сказанное выше вовсе не должно относиться к объектно-ориентированному методу RAD. Это ясно показал опыт работы автора в сфере финансовых услуг, где за распределением премий или построением кривых выручки кроются сложные расчеты и где была доказана применимость метода SOMA.

Метод DSDM предлагает подход к созданию прототипов, аналогичный нашему подходу к получению знаний в целом. Мы проводим широкий поверхностный анализ (создаем прототип), охватывая все основные свойства намеченной системы в общих чертах. За этим следует глубокий анализ (более подробный прототип) одной области задачи. Тут возможны три варианта.

1. Можно сначала приняться за самую легкую часть, чтобы вселить уверенность в разработчиков.
2. Сначала можно исследовать область, которая позволит разобраться с 80% потребностей.
3. Сначала можно взяться за технически наиболее сложную часть.

Первый подход редко бывает правильным, потому что впереди команду могут ожидать весьма неприятные неожиданности, да и просто нехорошо хвастаться перед пользователями, демонстрируя свое мастерство. Идеальной будет комбинация оставшихся двух. При использовании объектно-ориентированного подхода к анализу гораздо больше внимания нужно уделять поиску потенциально пригодных к повторному использованию компонентов в каждой следующей функциональной области. Кроме того, следует с самого начала ориентироваться на важнейшие требования, а не на качества, которые в идеале хотели бы получить пользователи (или разработчики).

Полезный вклад метода DSDM (впервые предложенного, как я понимаю, Деем Клеггом (Dai Clegg) из компании Oracle) состоит в определении приоритетов на семинарах требований. Метод DSDM классифицирует требования таким образом: необходимые, нужные, возможные и желательные (но не в этой версии). Сокращенно эту классификацию называют MoSCoW — по первым буквам названий категорий (Must have, Should have, Could have, Want to have).

Метод DSDM предлагает привлечение специальных консультантов по взаимодействию человека и компьютера. Я же в корне не согласен с разделением проектирования пользовательского интерфейса и всей системы в целом. Ведь невозможно разработать хороший интерфейс для системы с непродуманной архитектурой. Поэтому я предлагаю сначала удостовериться, что достаточное количество разработчиков имеют опыт создания пользовательского интерфейса.

И наконец, метод DSDM предлагает следующую классификацию моделей.

- Коммерческие прототипы
- Прототипы возможностей/прототипы проекта
- Прототипы производительности
- Прототипы применимости

Здесь отражено разделение итераций проектирования и реализации функциональных требований в жизненном цикле метода DSDM (рис. 9.7) и его несовместимость с более целостными объектно-ориентированными подходами, например, описанными далее в этой книге. Автор выделяет только исследовательские, пробные (или одноразовые) и эволюционные прототипы. Он считает, что неправильно отделять функциональные требования от нефункциональных.

9.5. Модели жизненного цикла процесса и продукта

Как уже говорилось, большинство литературных источников по процессам разработки программного обеспечения ориентировано на нахождение наилучшего варианта процесса. Подразумевается, что такой процесс существует и он один. Процесс MOSES позволяет понять, что в действительности мы имеем дело с двумя жизненными циклами. Такие наблюдения были вынесены из работы над системой оценивания программного обеспечения, где всегда было очевидно существование двух типов метрик. В метрике продукта можно определить, что произведено, а в метрике процесса — как осуществлялось производство. Аналогично можно выделить два жизненных цикла в разработке программного обеспечения. Жизненный цикл продукта демонстрирует, как продукт поставляется покупателям. Моделью этого договорного процесса является "водопад". Простейшая модель разработки продукта показывает, что сначала мы создаем условия для разработки, т.е. находим бюджет, а потом уже создаем что-то и запускаем в эксплуатацию. Тем не менее производственный процесс вовсе не обязательно соответствует каскадной модели, он может представлять собой сложную систему параллельных действий. Эта дихотомия изображена на рис. 9.7 Нам уже знакомы такие сложные, параллельные процессы в производстве. При работе над программным обеспечением наблюдается та же картина.

Эта новая перспектива несколько проливает свет на прения о процессах разработки программного обеспечения. Она позволяет нам совместить и согласовать каскадный и итеративный подходы, вместо того чтобы выбирать между ними. Кроме того, она объясняет, почему не стоит заменять "водопады" "спиралями": ведь "спирали" оказываются свернутыми "водопадами". Решение заключается в том, чтобы увидеть отдельный жизненный цикл процесса, который должен протекать параллельно (за исключением совсем уж простейших случаев). Он описывает действия, предпринимаемые для создания продукта, и их взаимосвязь. Все это связано больше с производством, нежели с поставкой; речь идет о работе, а не ее оплате.

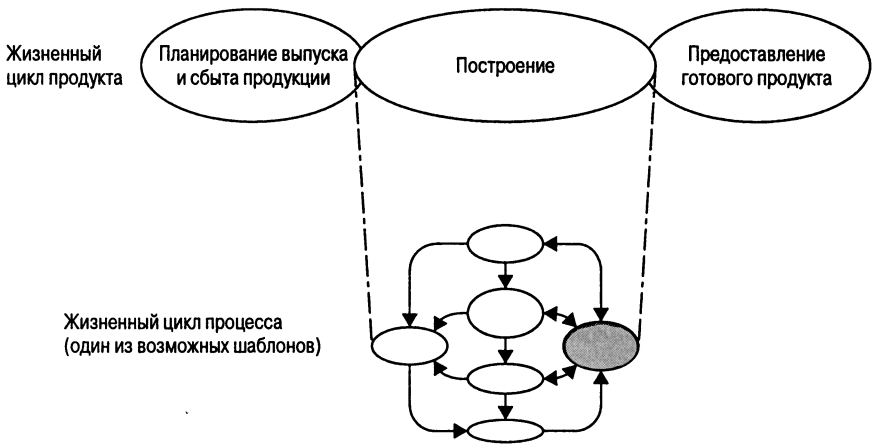


Рис. 9.7. Соотношение жизненных циклов продукта и процесса

Этот взгляд также позволяет совсем по-другому рассматривать сопровождение программного обеспечения (ПО). Традиционно, сопровождение ПО воспринимается как определенный процент от затрат на разработку, т.е. налог на использование программы. Очень редко на него специально выделяются средства.

На рис. 9.8 показано, что можно вообще упразднить само понятие сопровождения, если каждую доработку рассматривать как новый проект. Начальный жизненный цикл продукта здесь называется периодом роста программы, а последующие доработки представляются как замещающие продукты на стадии ее совершенствования. Каждый из этих продуктов проходит свой жизненный цикл, но повторяет один и тот же процесс и нуждается в отдельном бюджете. Все это тесно связано с идеями экстремального программирования и рефакторинга, хоть и представлено в более общем виде.

9.5.1. Объектно-ориентированные модели жизненных циклов

Известно несколько вариантов процессов для объектно-ориентированной разработки. Например, в [168] описан эволюционный процесс разработки на основе свойств, названный FFD (feature-driven development). Процесс FFD уделяет особое внимание тому, чтобы существенные результаты представлялись как можно чаще. У него много общего с процессом OPEN, в разработке которого был задействован Коад. Свойством называется небольшая часть функциональных возможностей программы, чем-либо полезных для клиента. Такую часть нужно предоставлять в готовом виде каждые две недели, а то и чаще. Этот метод включает в себя также планирование и использование шаблонов. По сравнению с классическими жизненными циклами, разработка FFD — это существенный шаг вперед, но все же этот процесс незначительно оторвался от спиральной модели. В нем почти совсем не учитывается тот факт, что на ранних стадиях готовыми компонентами могут быть еще не программы, а пока только четко сформулированные требования.

В [63] описан принцип экстремального программирования XP (eXtreme programming) — метод, требующий еще более частого представления существенных результатов. Технология XP основывается на тестировании, и простота здесь означает, что "в программе как раз всего

хватает, чтобы она прошла тестирование". Модульное тестирование проводится самими разработчиками, функциональное тестирование — пользователями. Результаты коротких циклов используются также для рефакторинга программы. Примером рефакторинга может служить создание суперкласса для выделения общих свойств, создание точек ветвления, разбиение классов или методов на две части и переименование компонентов для большей наглядности. Конечно, чтобы выполнять эти действия достаточно часто, необходимы автоматизированные средства тестирования.

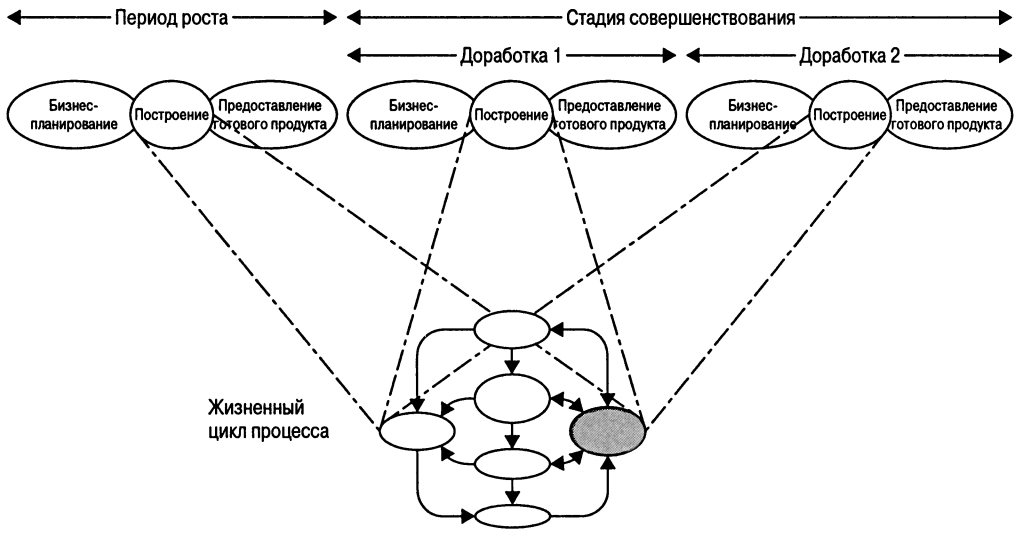


Рис. 9.8. Период роста и стадия совершенствования в жизненном цикле продукта, где каждый этап разработки включает жизненный цикл процесса

Парное программирование — это одна из наиболее важных и полезных методик XP (хотя придумали ее раньше). Эта методика заключается в том, что человек никогда не работает над программой один, т.е. пока один программист сидит и пишет программу, другой наблюдает, думает и комментирует. А потом они меняются местами. Более того, эти пары непостоянны, и каждый партнер может внести (и вносит) изменения в чью-то программу. Казалось бы, продуктивность такой работы должна уменьшиться вдвое, но, что любопытно, наблюдения показывают, что производительность повышается в два раза и более. Бек назвал этот подход экстремальным, потому что в нем хорошие общеизвестные идеи применяются "агрессивно". Например, он пишет следующее.

- Если рецензирование программы дает результат, выполняйте его постоянно (парное программирование).
- Если короткие итерации дают положительный результат, делайте их *действительно* короткими (длительностью по нескольку часов, а не месяцев).
- Если тестирование помогает, пусть все постоянно тестируется.
- Если вы стремитесь к простоте, создавайте простейшие работоспособные программы.

В методике XP подразумевается, что нужно постоянно следовать интересам дела (принцип рефакторизации). Тем не менее некоторые, особенно экстремальные, сторонники методики XP понимают это по-своему — мол, нет необходимости устанавливать требования, прежде чем писать программу: ведь главное, чтобы был какой-то смысл! Они считают, что если ты что-то неправильно поймешь, потом нетрудно будет изменить программу — так что не о чем беспокоиться. Я же думаю, что это неправильно по двум причинам. Во-первых, недопонимание требований может вылиться в серьезную проблему, когда команда разработчиков уже принялась за другие проекты. Во-вторых, я не верю утверждению Бека, что посредственные программисты сделают все как надо. Возьмите, например, случай с Крайслером (Chrysler) — широко известную историю успеха XP. Тогда в команде были весьма талантливые люди (и сам Бек в том числе). Более того, этот проект продолжил развитие ранее заваленного проекта, и большая часть старой команды была задействована в новой работе. Для меня неприемлемо предположение, что эти весьма опытные люди не имели четкого представления о требованиях еще до начала проекта. Я думаю, что технология XP — прекрасный способ разработки программ и все его методы полезны. И все же я считаю, что сопровождать его (не обязательно предшествовать ему) должны технологии разумного проектирования требований. И кроме того, он не подойдет компаниям, которые пытаются перейти от структурного подхода к объектно-ориентированным технологиям и переквалифицировать своих работников соответствующим образом; им и так нужно многому научиться, чтобы избежать, например, ошибок тех, кто одними из первых осваивал язык C++.

Если одним из важнейших основополагающих принципов объектных технологий является инкапсуляция, не может ли из этого следовать, что в нашем организованном подходе, в конце концов, может быть разрешена деятельность недисциплинированных хакеров? Раз объект определяется только его спецификацией и контрактом, его реализация может изменяться, не влияя на другие компоненты системы. Поэтому нам неважно, как реализуется объект, главное — соответствие спецификации и скорость. Вот тут-то и самое подходящее место для талантливого хакера. Ведь он может написать толковую, действующую программу. Ну, подумаешь, выглядит она бог знает как, в ней нет комментариев и сопровождать ее невозможно. Ну и не будем ее сопровождать; а понадобятся усовершенствования — создадим производный от нее класс. А не получится — выбросим ее и наймем другого хакера для выполнения работы. Таким образом, роли кустарного гения, т.е. хакера, выделяется законное место в индустрии разработки программ.

Некоторые организации на самом деле принимают приведенные доводы, и на это стоит обратить внимание. И все же, обычно далеко не всегда можно получить варианты программы всего лишь путем создания производных классов. Гораздо чаще нужно создавать абстрактный суперкласс путем обобщения программы. А для этого ее необходимо проанализировать и понять. Для любой большой организации это означает обязательное применение определенных стандартов, поскольку на каком-то этапе разработчикам придется разбираться в чьей-то программе. Похоже, что в этом отношении экстремальное программирование объединило лучшее из этих двух миров.

Кроме того, наш метод поддерживает так называемое хакерство или структурированную деятельность еще в некоторых аспектах. В рамках разработки предполагается, что осуществляется анализ, планирование и написание программы, но не указывается, в каком порядке выполняются эти действия. Это значит, что вполне допустимо планирование *после* написания программы. Это говорит о том, что действия по уяснению требований выполнены, но не о том, что требования полны. Анализ может следовать за написанием программы в процессе прохождения последовательных итераций по созданию прототипов. Тестирование и оценка

результатов временных блоков показывают, что такая практика действенна. Раз тестирование прошло успешно, не стоит беспокоиться о том, как был получен этот результат. Сам по себе временной блок имеет тенденции к ухудшению функциональности. Таким образом, мы установили на абсолютно неструктурированный созидательный процесс идеальную систему контроля и управления. И наши "хакеры" могут работать наиболее продуктивным, по их мнению, способом, лишь бы не страдало качество или сохранялась возможность сопровождения продукта.

Метод *Team Fusion* включает собственный, но легкий и приспосабливаемый для определенной цели процесс. Он строится на рисках и также подразумевает частый выпуск готовых компонентов. Как и при экстремальном программировании, главное — сделать "ровно столько, сколько нужно". Это был один из первых методов, содержащих идею шаблона процесса. Во всем остальном он основывается на довольно простенькой спиральной модели. Средство *Process Engineer* от компании CA строится на основе настраиваемого процесса, очень похожего и на метод *Team Fusion*, и на описанный в разделе 9.6 процесс, но с параллельными задачами оно справляется не очень хорошо. Данное средство включает множество технических приемов процесса Catalysis и уделяет много внимания компонентам. Процесс *Perspective* [27] также предлагает свой подход к управлению разработкой на основе компонентов, но он не силен в методах проектирования.

9.5.1. Технологии Objectory и Rational Unified Process

Уже в 1998 году в методе Objectory существовала модель жизненного цикла, основанная на прецедентах. Когда Якобсон (Jacobson) в 1995 году пришел в компанию Rational Inc., она приступила к продаже технологии Objectory и, по принятии группой OMG (рабочей группой по развитию стандартов объектного программирования Object Management Group) языка UML, переименовала ее в Rational Unified Process (RUP). То ли из-за видимой связи с языком UML, то ли из-за весьма агрессивного способа ведения торговли, RUP стал наиболее котируемым образцом объектно-ориентированного процесса разработки. В [463] процесс RUP описан в общих чертах, а в [416] подробно описана та его часть, которая называется USDP (Unified Software Development Process). В [670] рассматриваются некоторые аспекты процесса RUP с точки зрения руководства проектом. И хотя между изложенными в этих трех книгах взглядами довольно много противоречий, в этом кратком обзоре мы будем рассматривать все содержащиеся в них идеи как единое целое под одним названием — RUP.

Для каждого проекта метод RUP определяет виды деятельности, артефакты (результаты), исполнителей (роли в проекте), фазы и дисциплины. Эти термины, правда, несколько непривычны: например, большинство руководителей проектов называют дисциплины RUP *видами деятельности*. На самом деле, для этого есть стандарты ISO (Международной организации по стандартизации). Определения приводятся в табл. 9.2.

Обратите внимание на то, что все виды деятельности выполняются последовательно, хоть и итеративно; в этой модели параллелизм и синхронизация не предусмотрены. Взаимосвязь между фазами и дисциплинами показана на рис. 9.9, на котором также изображен жизненный цикл продукта в виде четырехступенчатого "водопада" (начало — развитие — конструирование — передача). Каждая ступень состоит из последовательных итераций, так что практически моделью процесса является обыкновенная спираль.

Таблица 9.2. Терминология RUP

Термин RUP	Обычный термин	Определение
Вид деятельности	Задача	Единица работы исполнителя по созданию готового артефакта
Этап вида деятельности	Подзадача	Часть вида деятельности, т.е. задачи
Артефакт	Готовый компонент	То, что производится людьми, выполняющими задачи. В RUP готовыми компонентами называются пригодные к использованию клиентом артефакты
Итерация	Итерация	Спланированная *последовательность* действий, завершающаяся выпуском готового компонента (или вехой)
Фаза	Этап/стадия	Период времени между основными вехами
Исполнитель	Роль	Обязанности разработчика, представленные в виде набора задач
Дисциплина	Вид деятельности	Последовательность действий, направленных на получение результата, имеющего измеримое значение для исполнителя. В методе USDP определяется диаграммой видов деятельности на языке UML

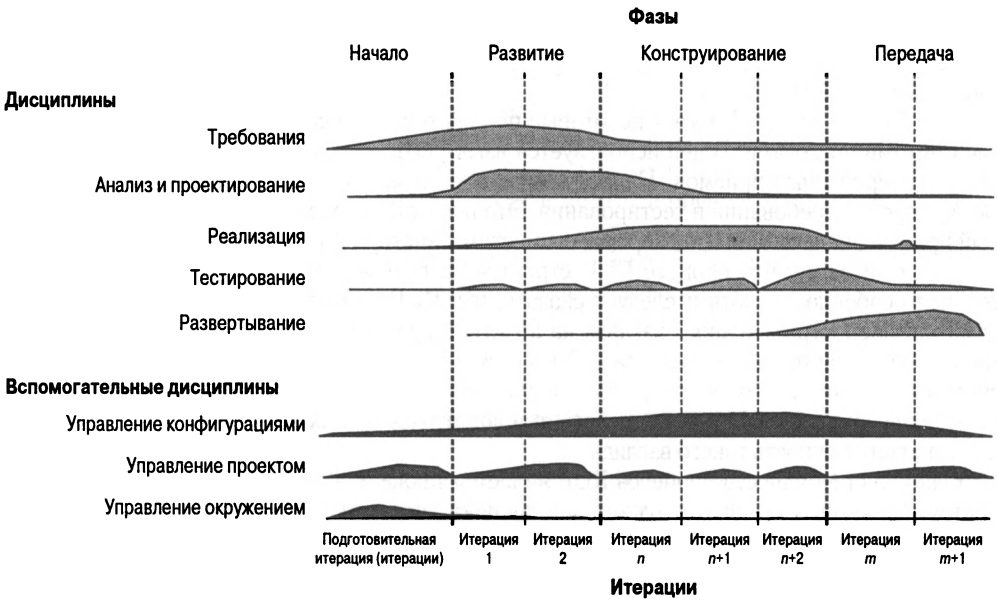


Рис. 9.9. Дисциплины и фазы процесса RUP

Обычно RUP-проекты осуществляются приблизительно в такой последовательности.

- Соберите команду.
- Решите, какую систему вы будете разрабатывать (кроме разработки системы, очевидно, вариантов нет).
- Постройте модель прецедентов и прототип пользовательского интерфейса.
- Используйте расширения процесса на языке UML для построения объектной модели анализа.
- Пользуясь обычными возможностями языка UML, разработайте модель проектирования — диаграммы классов, состояний, последовательностей и т.п.
- Хорошо обдумайте архитектуру, распределяя классы проектирования по модулям и пакетам.
- Проверьте модель прецедентов. Метод RUP предоставляет прекрасное руководство по проведению тестирования.
- Развивайте свой продукт до состояния рабочей системы.

Методика RUP допускает ограниченное количество организационных изменений. В RUP-разработке в рамках процесса можно менять исполнителей, виды деятельности и т.д. Но изменить последовательность более проблематично, а для описания параллельных процессов понадобится специальное расширение.

Процесс RUP включает также указания по использованию инструментальных средств Rational в процессе выполнения проекта. В версии USDP значительное внимание уделяется использованию расширений процесса на языке UML в рамках дисциплины анализа. Здесь вместо диаграмм классов на языке UML используется предварительная модель с обозначениями интерфейса, модуля и управляющих объектов. На опасность такого подхода я уже указывал неоднократно.

Технология RUP имеет некоторые положительные особенности по сравнению с более старыми подходами. В ней используется язык UML и несколько исключительно объектно-ориентированных приемов. Наиболее важным является использование прецедентов для формулировки требований и тестирования. Это в высшей степени итеративный и инкрементный процесс, и, кроме того, он хорошо совместим с инструментальными средствами Rational.

Считается, что процесс RUP строится на основе архитектуры. Это тоже положительная особенность, хотя и следует сказать, что RUP придерживается раскритикованного нами в главе 7 ограниченного взгляда на архитектуру как на полную структуру. Метод RUP также основывается на прецедентах. Считается, что прецеденты определяют спецификацию системы; однако ничего не говорится о коммерческих требованиях или моделировании бизнес-процессов (кроме того, что прецедентов достаточно для их описания). В главе 8 мы уже показали ошибочность такого взгляда.

Одно из преимуществ процесса RUP является также и его недостатком: привязанность к инструментальным средствам одного поставщика часто стесняет организации, которые не могут пользоваться новинками по мере их поступления. Я уже отмечал, что методику RUP нужно расширить с точки зрения инженерии требований. В главе 8 было показано, как это сделать при помощи микропроцесса. К тому же метод RUP не содержит никаких сведений по созданию графического интерфейса пользователя. И опять-таки, это легко исправить. Метод

RUP не содержит толковой информации и о средствах CBD, но вот технические приемы Catalysis и описанный в главах 6 и 7 микропроцесс можно смело добавлять. В технологии RUP нет четко определенной системы показателей, хотя предполагается, что их следует собирать. Но и тут тоже возможно расширение, которое будет описано в разделе 9.7. Пожалуй, худшая особенность технологии RUP как современного процесса — его размер, превышающий 1700 страниц. Легким такое не назовешь. Далее мы покажем, как технологию RUP можно расширить и обобщить при помощи истинно легкого и гибкого процесса.

9.5.2. Процесс OPEN

Начиная где-то с 1995 года консорциум OPEN вырос в неофициальную группу, которая включала приблизительно 30 методистов с разной коммерческой специализацией, стремящихся к большей методической интеграции и убежденных, что в методах должен заключаться процесс, они должны быть доступны всем, не должны привязываться к определенным инструментальным средствам и должны сосредоточиваться как на научной целостности, так и на прагматических вопросах. Основателями OPEN были Брайан Хендерсон-Селлерс и я, принявшиеся объединять модели процессов MOSES и SOMA. Результат — процесс OPEN — был опубликован в 1997 году [333]. Вскоре к нам присоединился Дон Файерсмит (Don Firesmith), начавший работу над интегрированной системой обозначений — языком моделирования объектов OML. Его цель —добиться более объектно-ориентированного характера, чем у языка UML, на который повлиял OMT, а также достичь большей легкости для изучения и запоминания [272]. Эта система обозначений, хоть и была основана на простых идеях, стала очень сложной, и, возможно, отчасти поэтому на нее обращали мало внимания после принятия языка UML группой OMG.

Процесс OPEN включает адаптируемый технологический процесс, основанный на объектной модели жизненных циклов продукта и процесса, где жизненным циклом продукта является "водопад", а процесса — "сеть", как было описано ранее. Консорциум тогда напечатал очередную книгу, рассказывающую, как разные методы инженерии программного обеспечения соотносятся с видами деятельности и задачами процесса [381]. А по-моему, четкой связи между всеми приемами в нашем арсенале быть не может, но этот труд является интересным примером того, как некая организация могла бы взяться за создание такой матрицы — это было бы достойным подтверждением идей. Главным новшеством этой работы стала разработка матрицы, демонстрирующей, как разные методы и приемы соотносятся с видами деятельности и задачами процесса. Например, метод диаграмм Ганта (Gantt) можно порекомендовать для планирования проекта. Более того, можно было бы определить методы как рекомендованные или обязательные для соответствующих задач. Это тоже будет объяснено в следующем разделе. Как мы сейчас увидим, процесс OPEN является настраиваемым во всем.

9.6. Модель процесса на основе контрактов

У большинства конструктивных объектно-ориентированных методов анализа и проектирования есть отличительная особенность: модели их жизненных циклов остались строго процедурными. Некоторые из них представляют собой не больше, чем просто "структурированные" методы с добавлением объектов. Процесс OMT вообще включает рудиментарный микропроцесс для анализа и разработки. Процесс MOSES был первым методом, который представил полный жизненный цикл, хотя в технологии Objectory был свой собственный.

Метод MOSES сохранил несколько процедурный оттенок, потому что в нем по-прежнему были фазы, хотя их, наверное, можно было бы выполнять в любом порядке. У других методов и вовсе не было моделей жизненных циклов, а если и были, то очень слабенькие [99]. Для успешной быстрой объектно-ориентированной разработки нужно полностью отказаться от понятия "фаз" разработки.

В модели жизненного цикла OPEN, основанной на контрактах или описаниях [334], проект определяется как система взаимосвязанных видов деятельности без четко установленного порядка выполнения. В нем различаются два жизненных цикла: жизненный цикл продукта в виде "водопада" и жизненный цикл процесса в виде "сети". В процессе OPEN итогом вида деятельности должен быть *протестированный* результат. В частности, задачи тестируются в процессе их выполнения, а объектные модели требований — в процессе сквозного контроля, как описано в главе 8. Сценарии прецедентов реализуются как можно раньше и потом используются для тестирования. Это помогает подходить к проблеме отладки независимо от инструментальных средств и подчеркивает первоочередную важность качества. Эта стратегия проверенных временны*х* блоков (или принцип "как раз вовремя") делает вполне возможным и даже безопасным написание программы до планирования (или наоборот — это уж как вам нравится работать) — прямо в духе технологии XP. И объекты предметной области, и сценарии решения задач собираются в архиве для повторного использования в последующих проектах и на будущих семинарах. Дело не в том, что объектная ориентация должна улучшить методику RAD или наоборот, а в том, что современные информационные технологии требуют согласованного, непротиворечивого применения преимуществ обеих технологий. Главное преимущество объединенного подхода состоит в том, что он позволяет перейти к повторному использованию не только на программном уровне, но и на уровне спецификаций.

В этом разделе описан мой подход к разработке программных систем. Он представляет разработку любой компьютерной программы как набор видов деятельности, каждому из которых отведен свой подраздел. Задачи, выделяемые при этом подходе, в настоящее время считаются лучшим примером разработки программ и управления проектами. Этот процесс уже успешно использовали такие организации, как Swiss Bank Corporation, Chase Manhattan Bank, и некоторые клиенты компании TriReme всегда с какими-то поправками. В этой главе освещается только модель процесса. А о соответствующих приемах объектно-ориентированного анализа и проектирования уже было немало сказано.

Разница в акцентах традиционной и объектно-ориентированной разработок заключается в том, что объектно-ориентированный процесс сглаживает различие между анализом и логическим проектированием, но разделяет проектирование на логическом и физическом уровнях. В нашем подходе есть лишь один крупный документ, проходящий три фазы разработки: отчет по семинарам, отчет по результатам анализа и отчет по результатам проектирования. Для каждого вида деятельности добавляются новые главы и изменяются некоторые из уже существующих. Тестирование приемлемости для пользователя UAT (User Acceptance Testing) равномерно распределяется и проводится по ходу проекта, а не после его завершения. В этом отношении особенно большое значение имеют рецензии пользователей (User Review), что, впрочем, не избавляет от необходимости тестирования UAT. Управление изменениями осуществляется, как обычно. А по разработке пользовательского интерфейса мы предлагаем специальные инструкции.

Определения

Проектом (project) называется любая работа, которая предположительно должна занять не менее пяти рабочих дней. Оптимальное время выполнения проекта — от одного до двенадцати человеко-месяцев. При наличии команды из двух–четырех человек такой проект будет выполнен не более чем за полгода. Работа, отнимающая менее пяти дней, называется запросом на предоставление услуг RFS (Request For Service), и руководство уделяет ей первостепенное внимание. Запрос на внесение изменений называется CR (Change Request) и обычно требует около одной человеко-недели.

Программа (programme) — это набор проектов, разработанный для выполнения крупной корпоративной задачи. Для применения описанного здесь подхода программы нужно делить на проекты. Уже само такое деление может вылиться в целый проект. Рекурсивная структура этого процесса изображена на рис. 9.10.

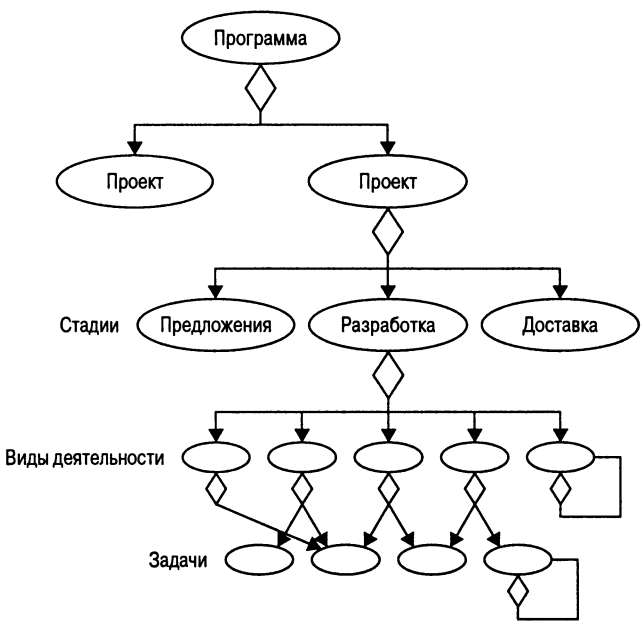


Рис. 9.10. Программы, проекты, стадии, виды деятельности и задачи

Для максимального увеличения эффективности предоставляйте пользователям возможности для тестирования результатов, а также ориентируйте команду разработчиков на необходимость уложиться в определенные сроки, ведь проекты по разработке выполняются в рамках временны*х* блоков. Ограниченный временны*м* блоком проект в идеале займет от 30 до 180 дней, и для этой работы потребуется от двух до шести человек. Если же проект невозможно ограничить такими рамками, так как требования слишком сложны, работа разбивается на части, соответствующие приведенным стандартам. Когда масштаб проекта настолько ограничен, что требуемые трудозатраты слишком малы, ряд требований объединяют для создания нормального проекта. Обратите внимание на то, что в таком случае все требования должны быть взаимосвязаны и ранжированы пользователями или другими участниками проекта.

Устранение сбоев в промышленной версии требует особого внимания: эта задача важнее работы над проектами и предоставления услуг.

Методики и порядок действий, описанные в этой книге, годятся только для тех проектов, которые требуют трудозатрат от 4 до 36 человеко-месяцев; к меньшим проектам следует применять сокращенную форму этого цикла. А об этом речь пойдет в разделе 9.6.16.

Модель на основе контрактов

В предлагаемом подходе нет линейной организации "технологических процессов" или "фаз", но есть виды деятельности, ограниченные и неограниченные. Ограниченные виды деятельности укладываются во временные блоки — четко оговоренные периоды затрачиваемого времени. Некоторые неограниченные виды деятельности продолжаются постоянно, но затраты на них тоже нужно контролировать, устанавливая цели и достигая их. Другие же входят в ограниченные виды деятельности, и руководитель проекта должен регулировать затраты средств и времени на них, исходя из отведенных на весь ограниченный вид деятельности ресурсов. Некоторые мелкие ограниченные виды деятельности соответствуют итерациям метода RUP.

Эта модель жизненного цикла продукта "навеяна" моделью процесса MOSES. Она состоит из трех стадий: предложение, разработка и реализация. Самая сложная из них — стадия разработки. Жизненный цикл процесса частично покрывает все три стадии, но большее отношение имеет к разработке.

Виды деятельности завершаются определенным результатом. Виды деятельности можно рассматривать как объекты с состоящим из задач интерфейсом. Разработчики и пользователи, выполняя свои роли, отвечают за решение задач и производство готовых компонентов. Для этого они используют такие **методы**, как "создание диаграммы последовательностей", "написание кода на языке Java" или "моделирование прецедентов". В этой главе методы не рассматриваются: мы считаем, что они уже знакомы читателю. Каждая зависимость между видами деятельности — это контролируемое сообщение, поскольку отправка каждого сообщения "контролируется" определенными тестами. Всякий такой контроль — это набор правил или формальное утверждение, описывающее семантические свойства элементов программного обеспечения (часто пред- или постусловие). Выполнение каждого правила обеспечивается тестом. Требуемый готовый компонент становится постусловием всего вида деятельности в целом, что подробнее описано в последующих подразделах. Виды деятельности — это строительные блоки проектов, соответствующие дисциплинам и фазам процесса RUP. Можно пойти дальше простого рисования диаграмм видов деятельности для их представления. Определение правил контроля означает создание сети связей между видами деятельности, которые могут выполняться параллельно. Для каждого из них проводится рецензирование и тестирование.

При этом применяются следующие принципы.

1. Качество превыше всего: все элементы тестируются по мере создания. В данном случае частично устраняется необходимость в рецензировании после реализации, и его вообще можно отменить по договоренности с клиентом.
2. Постоянное усовершенствование основывается на повторном использовании программного обеспечения и спецификаций и на измерении метрик продуктов и процессов.

3. Минимальные сроки продвижения продукта на рынок достигаются благодаря подходу RAD с использованием временных блоков.
4. Гибкость и устойчивость достигаются благодаря строгому следованию объектным и компонентным технологиям на протяжении всего жизненного цикла.

Высокоуровневая структура проекта

Высокоуровневая объектная модель проекта на основе контрактов изображена на рис. 9.11, где основные внутренние виды деятельности проекта изображены в белых эллипсах, а внешние — в серых. Обратите внимание на то, что эта модель жизненного цикла является подлинно объектной. Здесь виды деятельности являются объектами, а задачи — их "методами".

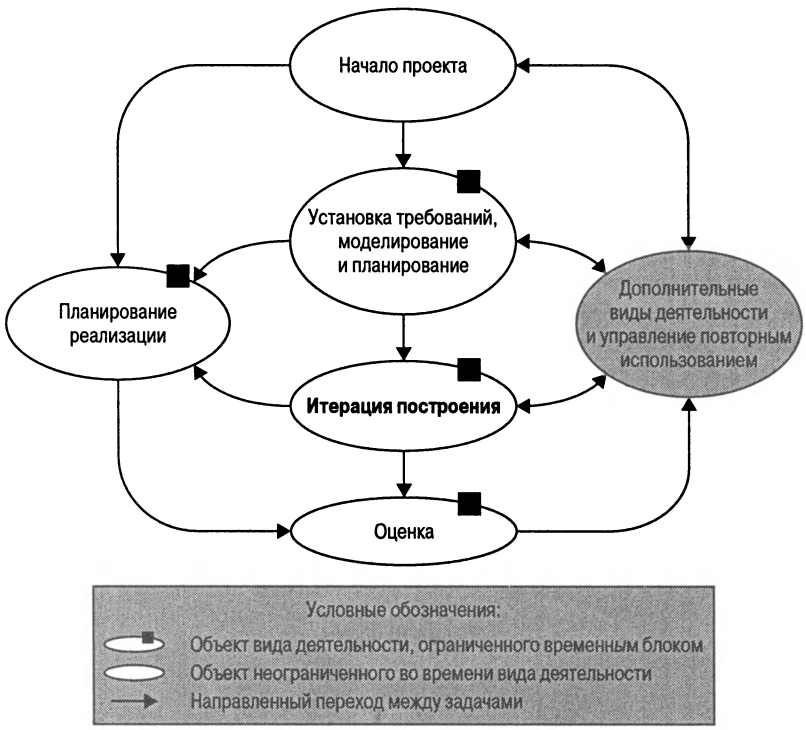


Рис. 9.11. Жизненный цикл на основе контрактов

Каждый эллипс на схеме представляет объект вида деятельности — это действия или прецеденты. Объекты, отмеченные черным квадратиком, ограничены определенными временными рамками. Для проверки результата каждого вида деятельности есть свои критерии. И в каждом виде деятельности есть предусловия, которые должны быть выполнены к тому времени, как начнется и/или завершится работа над какими-то его задачами. Виды деятельности также можно рассматривать как объекты, посылающие друг другу контролируемые (т.е. тестируемые) сообщения в виде готовых компонентов. В качестве правил контроля выступают формальные утверждения, описывающие семантические свойства элементов

программного обеспечения, или (что то же самое) пред- и постусловия видов деятельности или задач. Вид деятельности состоит из **задач**, соответствующих действиям объектов вида деятельности. Иногда формальные утверждения, описывающие семантические свойства элементов программного обеспечения, являются пред- и постусловиями индивидуальных задач. Для получения результатов в задачах используются методы.

Чтобы запустить проект в действие, нужно получить соответствующую санкцию всеми возможными способами: проведя анализ затрат и результатов или напоив руководителя джином! Постусловием станет подписанный документ о начале проекта с оговоренным бюджетом. Как только мы покончили с этим неограниченным видом деятельности, можно одновременно приступить к формулировке требований, развитию и планированию процессов в рамках проекта, планированию видов деятельности по реализации, а также начать необходимую деятельность вне самого проекта. Как только мы выработали требования, обычно в виде объектной бизнес-модели, можно приступать к итеративному виду деятельности по разработке. А параллельно можно продолжать планирование реализации. Сюда войдут все те скучные мелочи, без которых не видать успешного завершения проекта. Необходимо убедиться, что у каждого разработчика есть рабочее место, обеспечить набор инструментальных средств и т.д. И наконец, мы оцениваем результаты и решаем, выпускать ли этот продукт.

Прежде чем двинуться в путь...

Рассматривая прецедент определения требований, как показано на рис. 9.12, несложно заметить, что он начинается с выявления знаний — семинаров или опросов относительно требований. Естественно, мы предполагаем использование семинаров. По окончании семинара возникают задачи по разработке модели и анализу. Обратите внимание на то, как это проясняет связь между идеями процесса RUP о фазе развития и дисциплине анализа. Параллельно можно проводить другие семинары. Мы также считаем, что здесь начинается работа над задачами по планированию проекта и выделению итераций.

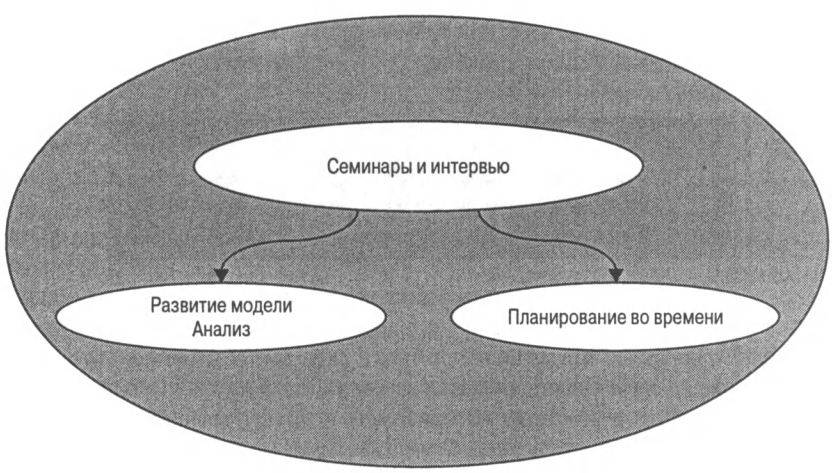


Рис. 9.12. Виды деятельности до начала разработки

Итерации разработки

Разработка состоит из коротких циклических итераций, как видно из замкнутой цепи сообщений, изображенной на рис. 9.13. Заметьте, что порядок выполнения анализа, разработки, написания программы и документирования не навязывается. В проекте, основанном на соглашении, важно лишь достичь цели конкретного вида деятельности: разработка должна быть хорошо задокументирована, программа проверена и т.д. Одним нравится писать программу еще до разработки. Другим удобно строить разработку на предварительно написанной документации. Не имеет значения, в какой последовательности вы выполняете работу. Я считаю, что людям нужно позволить работать так, как у них лучше всего получается, а не заставлять их следовать каким-то традициям. Пользователи часто рецензируют результаты — и это может привести к доработке требований. Переговоры о таких изменениях зависят от того, какие задачи были признаны первостепенными во время работы над требованиями. Если время, отведенное на создание проекта, истекло или продукт признан удовлетворительным, команда должна объединить свой продукт с результатами предшествующих или параллельных временны́х блоков, проверить его на соответствие стандартам, документации, провести тестирование и т.д. Она также может выделить пригодные к повторному использованию компоненты и передать их в библиотеку предметной области. Но писать пригодные к повторному использованию программы никто не заставляет. Такому сценарию прекрасно соответствует технология XP.

Рис. 9.13. Виды деятельности и итерации разработки

Программные виды деятельности

Проект связан с видами деятельности, не входящими в сам проект (рис. 9.14). Мы уже видели, что разработчики передают пригодные к повторному использованию компоненты специалистам, обслуживающим библиотеку. Несомненно, эти же специалисты будут поддерживать и повторное использование результатов. Ясно, что руководители программ и проектов должны сотрудничать и распределение ресурсов всей компании связано с распределением ресурсов проекта. Когда система запускается в эксплуатацию, часть команды занимается

отработкой и отладкой, работает с первыми пользователями и улаживает первоначальные трудности. Остальные участники проекта работают с группой, занимающейся повторным использованием, приводя свой потенциально пригодный к этому компонент в "товарный вид". Но это мы обсудим в разделе 9.7.

Рис. 9.14. Внешние по отношению к проекту (программные) виды деятельности

Полный жизненный цикл

Все возможные сообщения между различными видами деятельности изображены на рис. 9.15. На первый взгляд эта диаграмма может показаться слишком сложной, но если виды деятельности реализовать в виде карты изображений на языке HTML, тогда она придаст документу процесса действительно дружественный интерфейс. Мы включили ее в описание для полноты картины.

В такой модели процесса эволюционное создание прототипов и ускоренная разработка — правила, а не исключения. В результате таких видов деятельности, как начало проекта, формулировка требований и анализ, могут создаваться и одноразовые прототипы.

Руководство контролирует финансирование проектов на высоком уровне, и согласно его решениям создается план реализации проекта и проводятся семинары с целью установить желательные возможности приложения и цели всех категорий пользователей. Затем проводятся интенсивные семинары по уточнению требований, в результате которых создаются модели бизнес-процессов, сценарии задач и общая объектная модель, после чего следует более тщательное и критическое уточнение этой модели и создание отчета по результатам анализа. Результаты семинаров по уточнению требований и последовательный анализ влекут за собой необходимость планирования разработки и моделирования предметной области по двум причинам. Во-первых, планировщиков нужно ознакомить с требованиями к ресурсам, выставленными при первичном определении масштабов приложения. Во-вторых, "хранителям" модели предметной области нужно рассмотреть объектную бизнес-модель, чтобы поискать в своей базе подходящее описание класса. Разработчики, в свою очередь, черпают ресурсы проекта у планировщиков, а пригодные к повторному использованию спецификации и программы — у "хранителей" библиотеки.

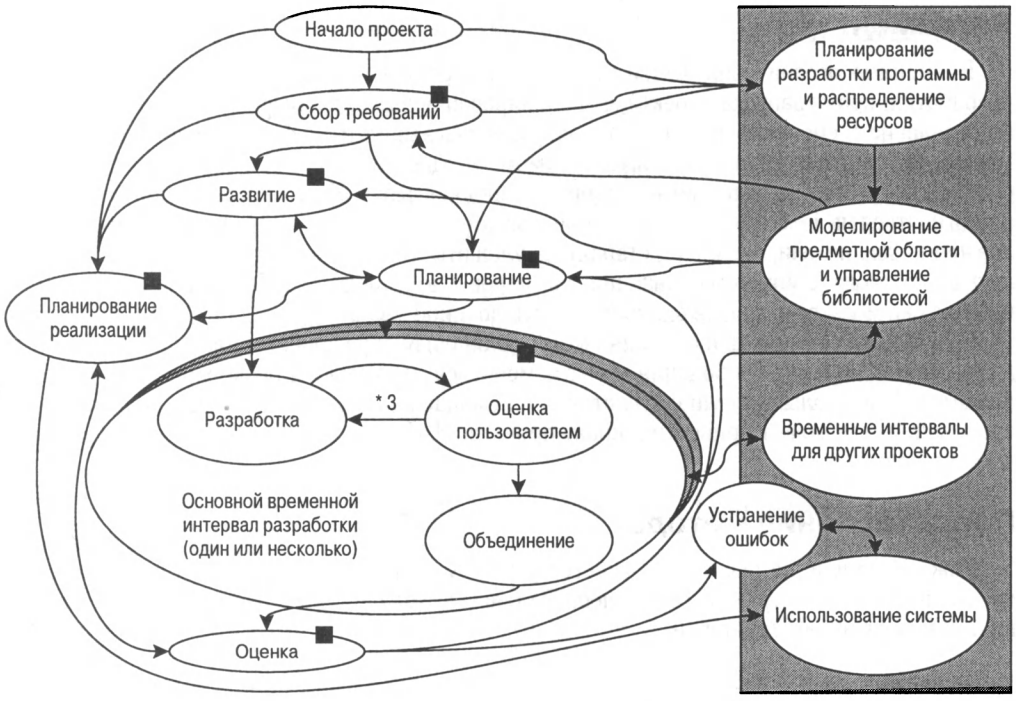


Рис. 9.15. Полная объектная модель проекта на основе контрактов

Выполнение каждого вида деятельности — это этап, по завершении которого получаются выходные данные и результаты. Выходные данные — это нечто неизбежное, которое никому, кроме разработчиков, не показывается. А результаты видят и другие люди. Их делят на опциональные, рекомендованные и обязательные, в зависимости от их важности. **Обязательные** (Mandatory deliverable) результаты нужно представить во что бы то ни стало. **Рекомендованные** (Recommended deliverable) можно и пропустить, если это оправдано. А **опциональные**, или **факультативные**, (Optional deliverable) руководитель проекта может отменять по своему усмотрению. Критерии завершенности всегда включают тестирование результатов видов деятельности. Таким образом, задачи и виды деятельности помечаются буквами О (обязательные), Р (рекомендованные) или Ф (факультативные). Эту схему можно рекурсивно применить к подэтапам, подзадачам и подвидам деятельности. Мы еще не воспользовались двумя другими деонтическими[1] уровнями из спецификации процесса OPEN: **не одобряется** и **запрещается**.

Предлагаемый метод настаивает на том, что разработка программы и восстановление бизнес-процесса — это одно и то же. На каждом семинаре по уточнению требований мы ищем возможности радикального обновления процесса и не связываем себя существующей функциональной организацией компании или ее отделов. Наш подход к объектному моделированию начинается с описания бизнес-процессов и задач, представленных в этой модели, но скрытых за разговорами. Модели крупных процессов представляются с помощью метода Mission Grid.

---

[1] Деонтика — это наука (или логика) долженствования или обязательства.

Регламент

Временные ограничения всегда устанавливаются для всех видов деятельности быстрой разработки, кроме запуска проекта, его планирования и моделирования предметной области. Определение требований и уточнение плана должны занимать не более четырех недель. Возможно, потребуется ограничиться минимумом — одной неделей. Планирование временных блоков также обычно занимает от одной недели до четырех. Каждый временной блок может растягиваться на 1—5 месяцев, но нормой считается три. Оценка не должна затягиваться более чем на две недели, а вообще, старайтесь уложиться в полдня. В целом все это должно занять до полугода, с момента окончания семинаров и до реализации. Самый короткий временной блок на моем веку длился одиннадцать с половиной часов.

Результаты семинаров и особенно отчет по анализу определяют и план проекта, и отводимое на него время. Требования могут изменяться, но только в оговоренных в это время пределах. Если пользователи не участвуют в семинарах, не дают рецензии и не исправляют требования, это может привести к краху проекта. Избежать этого можно, только подписав договор об отсрочке.

Представление метода

Каждый эллипс на рис. 9.15 соответствует виду деятельности. Все виды деятельности в модели на основе контрактов (соглашений) описываются в разделе документации со следующей фиксированной структурой.

```
Название вида деятельности
Его описание и обоснование
Предусловия
Задачи (с указанием уровня важности)
Результаты и постусловия (включая критерии завершенности и
временные рамки)
```

В представленном в этой главе описании процесса есть ссылки на множество методов, рассмотренных в других источниках. К руководству по использованию приспосабливаемого к целям каждой отдельной организации процесса должны прилагаться ссылки на его информационную стратегию, стандарты разработки реляционной базы данных, политику безопасности, сетевое решение и т.д. Если в рамках проекта создается графический пользовательский интерфейс, необходимо соответствующее руководство по стилю его оформления. Также следует придерживаться стандартов кодирования и программного интерфейса приложения, если они были разработаны специально для используемого языка или среды.

В разделе 9.7 вы можете ознакомиться с реализацией описанного здесь жизненного цикла процесса на основе соглашений. Если читателя не интересует сам процесс, можно пропустить или пролистать эту главу, по крайней мере, при первом прочтении книги.

9.7. Подробнее о процессе на основе соглашений

В этом разделе речь пойдет всего лишь об одном способе представления процесса. Настоящее практическое пособие по этому методу должно содержать различные приложения, представляющие собой контрольные таблицы, дополнительные виды деятельности, связанные с методом документы, специфические вопросы, обсуждение ролей в рамках проекта и т.п. В разделах 9.7.1—9.7.12 описываются виды деятельности по разработке ядра системы. В разделах 9.7.13—9.7.16 говорится о задачах обеспечения управления проектом, качества и безопасности, которые идут вразрез с основными видами деятельности, наряду с исключениями, которые прилагаются в таких случаях в виде маленьких проектов. Во всех этих разделах мы будем обозначать общую организацию разработки как ИТ (информационная технология), а обобщенных заинтересованных лиц будем называть "пользователи". В последнем разделе перечислены роли и обязанности участников проекта.

9.7.1. Стадия начала проекта и связанные с ней виды деятельности

В этом разделе речь пойдет о той стадии, когда проект предлагается для рассмотрения и начинается его реализация. Эта стадия обязательна. Она соответствует начальной фазе процесса RUP.

Процесс разработки начинается с составления бизнес-плана. Этот процесс может быть вызван возникновением новых требований или даже внешним законопроектом. Результатом является предложение проекта, которое должно быть коротким и по существу. Оно должно содержать определение целей проекта, затраты и другие ограничивающие условия.

Проект начинается с получения утвержденной и подписанной в пользовательском отделе титульной страницы проекта. Именно на этом отделе лежит ответственность за подписание проекта, который должен содержать такие пункты.

- Постановка задачи, описанная в терминах бизнес-процесса, и ее коммерческое значение
- Имена спонсоров, экспертов в данной области и главных пользователей
- Обзор возможностей проекта и его факторы риска
- Обоснование в терминах, принятых организацией
- Критерии завершения и главные составляющие

Кроме того, проект может содержать расчет затрат, выполнимости, риска и критических значений различных факторов, от которых зависит успех. Он может сопровождаться прототипом, обеспечивающим "проверку концепции". При желании можно использовать стандартную форму запроса на проект. Обычно предпочтительнее подавать документы электронной почтой. Когда предложение подготовлено и представлено соответствующей комиссии для определения его порядка срочности, его помещают в книгу проектов, и можно начинать разработку.

Утвержденное и согласованное предложение переходит в стадию разработки на семинарах по требованиям — альтернативой может быть интервью всех пользователей. На этой стадии не предполагается, что результат будет внесен в компьютер и на семинарах можно свободно пользоваться всеми возможностями перестройки бизнес-процесса.

Предусловия

На этой стадии нет никаких предварительных условий.

Задачи

	ИТ	Пользователи	О/Р/Ф
Постановка задачи в терминах предметной области	Помогает	+	О
Определение границ — включение или исключение специальных вопросов или процессов	Помогает	+	О
Обоснование дальнейшего исследования	Помогает	+	О
Определение спонсора, желающего взять на себя ответственность за проект и отвечать за его успех или провал	Помогает	+	О
Описание критериев завершения	Помогает	+	О
Оценка (на высоком уровне) размера проекта в человеко-днях и ключевых задач, где это возможно	+		Р
Оценка затрат следующего вида деятельности проекта и затрат на весь проект, включая затраты на дальнейшее использование	+		О
Определение ресурсов и ролей для семинаров	+	+	Р
Определение требований к обучению персонала, проводящего семинары и обучение при необходимости	+		Р
Оценка существующих программ с целью найти существующие компоненты, пригодные для повторного использования, а особенно существующие модели	+		Р
Составление расписания семинаров и получение согласия участников	+	+	Р
Составление расписания специальных заседаний для частично занятых участников	+	+	Ф
Получение подписи и разрешения на запуск проекта от спонсоров и руководства	+	+	О
Начало проведения семинаров	+		Р
Начало предварительного планирования разработки, реализации и временных блоков. Поддержка связи с производством	+		Р

Результаты и постусловия

Главный компонент — это **Предложение по проекту**; документ, сгенерированный пользователями и ИТ-специалистами. Этот документ может занимать всего одну страницу; в этом случае его называют **Титульная страница проекта.** Для проектов большего объема используется второй документ, называемый **Качественным планом проекта**, в котором фиксируется прогресс по ходу выполнения различных этапов работы, в терминах ожидаемых и реальных сроков, и обязанности. Тогда руководство распределяет приоритеты проекта.

Рекомендуется, чтобы документ "Предложение по проекту" содержал следующие разделы.

1. Название проекта.
2. Требования в терминах предметной области и постановка задачи
3. Рамки проекта
4. Обоснование и окупаемость. Когда обоснованием служит возврат затраченных денег, для оценки движения денежной наличности используйте метод NPV
5. Спонсор-исполнитель, спонсор проекта и зависимости (люди, чья деятельность может повлиять на успех проекта)
6. Предположения и исключения
7. Внешние ограничения
8. Критерии завершенности
9. Планируемые затраты на работу, включая затраты на будущее использование (в том числе на аппаратные средства и программное обеспечение)
10. План обучения персонала, который будет проводить семинары
11. Оценочный размер программы с указанием ключевых задач и/или функций
12. Для больших проектов — план обеспечения качества
13. Словарь терминов

Это предложение должно быть полным, но занимать должно не больше одной страницы. Оно должно быть утверждено спонсорами и соответствующим руководством. Обоснование должно быть разумным и должно быть представлено в форме, принятой соответствующим главным управлением. Все необходимые и рекомендованные задачи (перечисленные выше) должны быть описаны полно; или должно быть обоснование того, почему какие-либо из рекомендованных задач не включены. Для больших проектов, кроме этого, должен быть представлен план обеспечения качества, который тоже должен быть одобрен руководством.

Это неограниченный вид деятельности, выполняемый безо всякого регламента.

9.7.2. Определение требований

Определение требований может послужить поводом для проведения совместных семинаров пользователей и разработчиков или более традиционных опросов, в зависимости от обстоятельств. Я настаиваю на проведении семинаров всякий раз, когда это возможно. И здесь, я полагаю, будут использоваться именно семинары.

Семинар метода SOMA — это шаблон процесса, разработанный для повышения качества и эффективности анализа систем в рамках жизненного цикла проекта. Команда экспертов

данной предметной области и специалистов по разработке программного обеспечения посещает семинары, чтобы определить требования к системе. Вырабатывается модель бизнес-процессов, определяются задачи пользователей, модель бизнес-объектов и отношений между ними. За этим следует уточнение плана построения объектной модели предметной области с использованием строгих методов объектно-ориентированного анализа. Там, где это подходит, на стадии уточнения плана выполняется модульная оценка, результатом которой является построение списка соответствующих модулей. Здесь требуется интенсивное привлечение пользователей. Семинары чрезвычайно выгодны, и количество примеров, подтверждающих это на практике, растет.

Хотя в качестве эволюционной методики реализации итераций для видов деятельности, ограниченных жесткими временными рамками, используются прототипы, необходимо понимать, что построенный на семинаре прототип почти со стопроцентной гарантией будет отброшен. Вывод: надо уметь правильно относиться к ожиданиям.

Определение границ системы

На семинарах по определению границ системы (или на отдельных заседаниях в рамках семинаров) определяется главное назначение проекта или его устав, границы рассматриваемой проблемы, вопросы, которые следует решить, задачи, приоритеты и цели пользователей. Кроме того, следует начать обзор бизнес-процессов. Определение рамок системы также подразумевает разбиение больших программ на проекты.

Результатом является документ, определяющий рамки системы. Он должен содержать описание назначения проекта, цели, метрики и приоритеты. Кроме того, он должен определять внешние объекты и главные роли, при помощи которых будет осуществляться взаимодействие пользователей с программой и друг с другом, а также события, которые будут вызывать эти взаимодействия и обмен информацией. Для каждого взаимодействия должны быть установлены постусловия. Предположения, исключения и ключевые вопросы документируются.

Сначала рассмотрим семинар, на котором определяются рамки проекта. За ним следует еще один или несколько семинаров, на которых оговариваются детали. О них речь пойдет позже, хотя они могут следовать непосредственно за заседанием по определению границ. Один из альтернативных вариантов — когда виды деятельности по определению границ и уточнению деталей объединяются.

Цели этого вида деятельности состоят в следующем.

- Достичь высокого уровня понимания существующих и предлагаемых бизнес-процессов, области их действия, выделить участников происходящих событий, посылаемые сообщения и их постусловия
- Определить назначение, рамки и цели проекта
- Установить четкие цели, приоритеты и критерии приемлемости для новой программы
- Рассмотреть финансовое и деловое обоснование и определить другие преимущества
- Завершить вид деятельности в короткие сроки, интенсивно используя семинары

Предусловия

Стадия начала проекта, описанная в разделе 9.7.1, должна быть завершена и одобрена соответствующим образом. Все инструментальные средства, т.е. текстовый процессор, графический пакет и т.д., должны быть доступны.

Задачи

	ИТ	Пользователи	О/Р/Ф
Систематизировать средства и возможности	+		О
Подтвердить работоспособность	+	+	О
Собрать важные документы	+	Помогают	О
Собрать спецификации объектов из архива и других проектов	+		Р
Подготовить план	+	Помогают	Р
Провести семинар по определению рамок системы	+	+	Р
Объяснить участникам подход к работе, включая обязанности спонсоров и пользователей	+		Ф
Определить и согласовать назначение, цели, методы оценивания и приоритеты	+	+	О
Зафиксировать предположения и исключения	+	+	О
Определить открытые вопросы и назначить ответственных за их решение	+	+	О
Создать модель бизнес-процесса	+	+	О
Определить все значительные взаимодействия	+	+	О
Установить постусловия каждого взаимодействия	+	+	О
Согласовать приоритеты реализации	+	+	О
Подписать соглашения и модели	+	+	О
Создать и напечатать документ, определяющий рамки системы	+		О
Пересмотреть и утвердить этот документ	+	+	О
Начать проведение семинара по уточнению деталей	+	+	О

Результаты и постусловия

Этот вид деятельности считается завершенным, когда существует **отчет о семинаре по определению рамок системы** и получены подписи пользователей. Цели тестируются при помощи попыток применения к ним метрик или посредством голосования. Назначение и цели тестируются до достижения консенсуса. Отчет должен содержать следующее.

1. Оглавление
2. Резюме (Р)

3. Введение и обзор проекта (Ф)
4. Название проекта и его назначение (О)
5. Список участников и даты проведения семинаров (О)
6. Цели, нормы и приоритеты (О)
7. Предположения и исключения (О)
8. Открытые вопросы и ответственные за их решение (Р)
9. Модель бизнес-процесса с указанием внешних и внутренних агентов и сообщений, которыми они обмениваются (О)
10. Замечания по проектированию и описание бизнес-процессов, включая вопросы управления и безопасности, устранения неполадок, описание транзакций с их объемами и частотой, ожидаемое время выполнения, главные интерфейсы других программ, принадлежность объектов (Р)
11. Подпись пользователей (О)

Пересмотрите документ, определяющий рамки проекта, чтобы убедиться, что он четко выражает следующее.

- Рамки проекта
- Цели проекта
- Рассмотрение возможности повторного использования кода программы, информационной модели и бизнес-модели
- Модель предложенного бизнес-процесса, его участников и взаимодействия
- Постусловия каждого взаимодействия

Семинар по определению рамок проекта обычно длится один день или даже меньше.

Семинары по уточнению деталей

Рассматриваемые семинары следуют за семинарами по определению границ и служат для усовершенствования модели бизнес-процесса и превращения ее в объектную модель задачи путем определения задач, необходимых для достижения каждой цели или постусловий. Выполняется декомпозиция этих задач, их классификация и определяется связь с ожиданиями. Текстуальный анализ сценариев задач приводит к созданию объектной модели предметной области, которая состоит из бизнес-объектов и их взаимоотношений.

Цели этого вида деятельности состоят в следующем.

- Определить требования к программе в деталях
- Определить набор сценариев действий по выполнению задач (прецедентов)
- Определить возможности нововведений в процесс
- Разработать объектную модель предметной области (в виде диаграмм(ы) классов на языке UML)

- Протестировать объектную модель предметной области на наличие прецедентов
- Дать рекомендации относительно необходимых ресурсов и оборудования
- Откорректировать планы разработки и реализации

На семинаре будут также рассматриваться вопросы соответствия, законности и безопасности.

Предусловия

Вид деятельности по определению рамок системы (если он применим) завершен, и соответствующий документ подписан спонсором и другими участниками.

Задачи

	ИТ	Пользователи	О/Р/Ф
Систематизировать средства и возможности	+		О
Подтвердить достижимость	+	+	О
Собрать важные документы	+	Помогают	О
Собрать спецификации объектов, пригодных для повторного использования, из архива и других проектов	+		О
Подготовить план	+	Помогают	Р
Объяснить участникам подход к работе, включая обязанности спонсоров и пользователей	+		Ф
Одобрить результаты семинара по определению границ	+	+	О
Подтвердить назначение и цели проекта	+	+	О
Подтвердить предположения и исключения	+	+	О
Подтвердить существование открытых вопросов и назначить ответственных за их решение	+	+	О
Подтвердить модель бизнес-процесса	+	+	О
Разработать сценарий задачи для каждого взаимодействия	+	+	О
Использовать модель процесса в качестве основы для определения и моделирования возможных модификаций процесса	+	+	Р
Разложить на элементарные задачи (прецеденты)	+	+	О
Выполнить текстуальный анализ сценариев для создания диаграмм классов	+	+	О

	ИТ	Пользователи	О/Р/Ф
Добавить архивные классы к разработанной модели	+	+	Р
Добавить к модели структуры	+	+	О
Определить компоненты и оболочки	+	+	Ф
Пройтись по всем значительным сценариям задач, чтобы создать диаграмму последовательностей	+	+	О
Создать матрицу взаимосвязей и целей	+	+	Ф
Согласовать приоритеты реализации	+	+	О
Подписать соглашения и модели	+	+	О
Создать и напечатать документ "Определение требований"	+		О
Согласовать документ "Определение требований"	+	+	О
Вычислить значения метрик	+	+	Р
Начать уточнение плана	+	+	Р
Начать планирование временны*х* блоков или подать информацию о нем	+	+	Р

Модели после этих семинаров следует распечатать, пересмотреть и согласовать со всеми участниками (если возможно, за один день).

Результаты и постусловия

Отчет о семинаре включает следующее.

1. Оглавление
2. Введение и обзор проекта (Ф)
3. Резюме по организации (Р)
4. Название проекта и его назначение (О)
5. Список участников и дата семинара (О)
6. Цели, метрики и приоритеты (О)
7. Предположения и исключения (О)
8. Открытые вопросы и ответственные за их решение (Р)
9. Модель бизнес-процесса с указанием внешних и внутренних исполнителей и сообщений, которыми они обмениваются (О). Сценарий задания вместе с его разложением на элементарные составляющие для каждого взаимодействия (О)
10. Объектная модель предметной области, состоящая из полных карт классов (О), диаграмм классов на языке UML (Р) и моделей состояний (О)

11. Набор диаграмм последовательностей, документирующих сквозной контроль (О)
12. План повторного использования результатов предыдущих семинаров, включая бизнес-модели, определения и разработки, содержащие списки объектов из предыдущих проектов, которые можно использовать повторно
13. Замечания по разработке и определения бизнес-процесса, включая вопросы безопасности и управления, устранения неполадок, описание транзакций с их объемами и частотой, ожидаемое время выполнения, главные интерфейсы других программ, принадлежность объектов (Р)
14. Следующие шаги и рекомендации (в том числе и по усовершенствованию процесса) (Р)
15. Словарь терминов и другая информация об этой области (Р)
16. Подписи пользователей (О)

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть обоснование того, почему какие-либо из рекомендованных задач не были включены.

Главный из применяемых тестов — это сквозной контроль, который выполняется при помощи ролевых игр и проверяет в основном объектную модель процесса и объектную модель задачи. Цели проверяются с помощью метрик и определения приоритетов.

Обязательно следует проверить, выполняется ли политика компании по информационной безопасности и классификации. Каждый объект в объектной модели процесса должен иметь своего владельца.

Вид деятельности считается завершенным, когда готов отчет и получены подписи пользователей.

Документ, описывающий требования, должен содержать:

- четкое определение всех требований и согласованные модификации процесса;
- полную объектную модель задач предложенного бизнес-процесса с разложением задач участников процесса на элементарные сценарии;
- постусловия каждого взаимодействия;
- полную объектную модель процесса, в том числе описания классов и структур;
- набор диаграмм последовательностей, представляющих ключевые события в данной области;
- набор метрик для модели;
- список пригодных для повторного использования классов из других проектов и предварительных кандидатов для повторного использования в проекте.

Кроме того, данный проект не должен быть двусмысленным и противоречивым. Семинар по уточнению деталей обычно длится до пяти дней и может включать определение рамок проекта. Для больших проектов может потребоваться больше семинаров. Официальный регламент — от одного дня до четырех недель. Эту задачу можно решить и за день, только руководителю разработки нужно представить формальное обоснование этих сроков.

9.7.3. Анализ и уточнение плана

Анализ состоит из двух этапов: сбор информации и моделирование, за которым следует уточнение модели. Обычно первый этап завершается во время проведения семинара, результатом которого является отчет, содержащий модель задач пользователей, сценарии выполнения задач и предварительную, но проверенную объектную модель бизнес-процесса, состоящую из полных карт классов и диаграмм последовательностей. К тому времени, когда начнется этот вид деятельности, обзор бизнес-процессов должен быть завершен. Каждое постусловие при определении рамок проекта разбивают на интересующие пользователей задачи. К ним относится изучение свойств программы, интерфейса и способа их реализации. Это важное дополнение к предложениям Якобсона [417], который стремится сделать описанные задачи (прецеденты) доступными только для исполнителей. Задачи объединяются в полную модель бизнес-процесса, и в отчет должны входить диаграммы задач с включением, где это нужно, всех четырех типов диаграмм классов на языке UML.

Теперь разработчикам нужно проверить модель на целостность и связность и попробовать найти какие-либо логические зависимости или неосуществимые аспекты. Таким образом, модели из отчета по семинарам усовершенствуются и образуют отчет по результатам анализа (OA), который, кроме всего прочего, должен включать планы и оценки проекта. OA дополняет отчет по семинарам и занимает его место.

OA должен содержать объектную модель задач и объектную модель бизнес-процесса, включая сценарии выполнения заданий, карты классов, диаграммы последовательностей, приоритеты компонентов и множество других материалов, которые были в отчете по семинарам, с дополнительными определениями, основанными на всестороннем и технически грамотном анализе проблемы, на который не всегда остается время при сумятице семинаров. Сценарии задач или прецедентов важны, так как они формируют основу сценариев тестирования. Во время подготовки OA могут приниматься предварительные решения о повторном использовании архивных объектов и интерфейсов существующих программ, и это также следует включить в отчет. Существует ряд аргументов в пользу применения шаблона обработки текстов для OA: он поддерживает более быстрое создание документов и их качество должно быть более высоким, поскольку информация подается только один раз и в нужном контексте. Это должно ускорить и облегчить проверку; рецензирование таких документов будет более простым, так как рецензент будет лучше представлять себе, что должно входить в каждый раздел, и это должно упростить доступ к информации для будущих разработчиков системы. Кроме того, упростится рецензирование документации программы внешними аудиторами.

Для соответствующих классов можно разработать диаграммы состояний на языке UML. Этот вид деятельности преследует следующие цели.

- Изучить альтернативные решения и выбрать самый лучший подход
- Построить прототип, где это нужно
- Подтвердить работоспособность объектной модели задач и записать постусловия для каждого действия
- Определить изменения в бизнес-процессе
- Создать модель успешных решений на языке UML для подтверждения технического подхода и определить критические области реализации программы

- Начать планирование возможностей, консультируясь с клиентами
- Рассмотреть соответствие политике безопасности
- Пересмотреть планы по разработке и реализации
- Добиться, чтобы пользователь подписал пересмотренную модель

Этот вид деятельности может выполняться в рамках временного блока или предшествовать ему. В простейшем случае он может полностью выполняться на семинаре.

Предусловия

Должны быть выполнены виды деятельности по инициированию проекта и начальному определению требований. Эти виды деятельности частично могут входить в состав самого семинара, но обычно выходят за его рамки и будут повторяться в рамках основного временного блока разработки.

Задачи

	ИТ	Пользователи	О/Р/Ф
Пересмотреть существующие программы и определить существующие компоненты для возможного повторного использования, особенно существующие модели	+		О
Решить нерешенные вопросы	+	+	Р
Подтвердить рамки и цели системы	+	+	О
Составить план развития продукта, консультируясь с клиентом	+		Р
Создать и напечатать отчет по результатам анализа, основанный на отчете по семинарам	+		О
Изучить ситуацию	+	+	Р
Подписать отчет по результатам анализа	+	+	О
Передать результаты в отдел по планированию временных блоков и по безопасности	+		О

Если в ходе анализа пакет решений утверждается, то следует выполнить следующий набор задач по оценке.

	ИТ	Пользователи	О/Р/Ф
Определить критерии оценки	+	+	О
Составить приглашение на тендер (ПТ)	+	+	О
Составить краткий список пакетов	+	+	О
Послать ПТ производителям пакетов			Р

	ИТ	Пользователи	О/Р/Ф
Представить подробную оценку пакетов из краткого списка	+	+	О
Оценить возможности аудита системы			О

Результаты и постусловия

Главная составляющая этого вида деятельности — **отчет по результатам анализа (ОА)**, который дополняет и улучшает отчет по семинарам. Его задача — построить модель, дать спецификацию предлагаемой программы в подробностях, рассчитать ее затраты, определить преимущества и сделать возможным принятие решения о том, стоит ли приступать к следующему виду деятельности. Отчет пишет команда разработчиков проекта. Его полное содержание здесь не приводится. Но, в любом случае, он должен содержать следующие элементы.

- Последний вариант всего того, о чем шла речь в отчете по семинарам
- Обзор влияния факторов риска системы
- Полную UML-модель предлагаемой программы, обязательно включая модели состояний
- Полный набор сценариев задач с пред- и постусловиями и необходимое количество диаграмм последовательностей
- Предварительный список компонентов, которые будут использоваться повторно
- Анализ влияния изменений на разрабатываемую программу
- План разработки, включая предположительную дату окончания работ
- Оценку метрик программы, включая затраты, не касающиеся непосредственно разработки

Должны быть выполнены все обязательные и рекомендованные задачи (перечислены выше) или должно быть представлено обоснование, почему какие-либо из рекомендованных задач не включены.

Официальный регламент — от одного дня до двух недель. Можно использовать общие временные рамки для проведения семинаров и выполнения анализа, составляющие от одного дня до четырех недель. Можно ожидать, что для комбинации этих видов деятельности минимальное время будет составлять одну неделю. Время, затрачиваемое на этот вид деятельности, включается в общий срок реализации (шесть месяцев).

9.7.4. Планирование временных блоков

План временных блоков дает общую схему видов деятельности и временных ресурсов, необходимых для создания согласованной части программы к установленной дате. Планирование временных блоков также включает в себя оценку и формирование ресурсов и инфраструктуры.

Этот план учитывается в общем плане проекта и посылается ответственному за составление общего плана.

Когда планируются параллельные временны*е* блоки или существует тесная взаимосвязь между результатами параллельных проектов, назначается дополнительное время для координации между такими временны*ми* блоками. Подробнее об этом речь пойдет в разделе 9.10.

Предусловия

К началу этого вида деятельности должна быть завершена стадия инициирования проекта. Формулировка требований и составление ОА (кроме отчета по временным блокам) должны быть завершены до окончания этого вида деятельности.

Задачи

	ИТ	Клиент	О/Р/Ф
Пересмотреть требования, сформулированные на семинарах, а также ОА	+		Ф
Пересмотреть модель предметной области и существующие программы и определить существующие компоненты, пригодные для повторного использования, особенно существующие модели	+		Р
Обеспечить запланированную программу обучения участников	+	+	Р
Применить средства проверки исходного кода к существующей программе, если предлагается вносить в нее изменения	+		О
Откорректировать планы разработки (с точностью до 30%)	+		Р
Определить влияние предлагаемых изменений на оборудование, включая пакетное и интерактивное времена отклика	+		О
Установить цели временны*х* блоков и опубликовать их	+		О
Установить число планируемых итераций для создания прототипа	+		Р
Утвердить команду разработчиков. Согласовать состав участников проекта с основными пользователями, руководителями проекта, разработчиками и спонсорами	+	+	О
Утвердить команду приемки результатов — спонсор, руководитель проекта, представители пользователей, посредник, демонстратор/читатель, аудитор, представитель производственного процесса, хранитель библиотеки повторного использования, юрист и т.д.	+	+	О
Закончить отчет о требованиях к обучению	+	Помогает	О
Отправить план руководителю разработки и проинформировать руководство об изменении предельных сроков и требований	+		О
Начать реализацию проекта	+		О

Результаты и постусловия

Результатом этого вида деятельности является **план временны*х* блоков**, который содержит диаграмму Ганта, и оценка ключевых задач или функций в человеко-днях с указанием необходимых ресурсов и сроков завершения. **Отчет о требованиях к обучению участников** делается для того, чтобы убедиться, что роли, обучение работе с программой и потребности в обучении правильно поняты и спланированы. Особое внимание уделяется требованиям и тестированию. За создание обоих отчетов отвечает руководитель проекта. Содержание отчета о требованиях к обучению будет таким.

1. Роли проекта и соответствующие имена исполнителей (Р)
2. Требования к обучению (Р)
3. График обучения (план высокого уровня) (Р)
4. Затраты на обучение (Р)

Отчет о требованиях к обучению помогает определить, какие умения требуются для успешного завершения проекта, а затем оценить навыки отдельных людей, чтобы выявить несоответствие. Убедитесь, что

- все члены коллектива учтены;
- определены требования к обучению и составлен его график;
- технические и деловые потребности удовлетворены;
- инспектирование включено в план.

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены; или должно быть обосновано, почему какие-либо из рекомендованных задач не включены.

Можно потребовать одобрения этого документа со стороны руководителя разработки и спонсора.

Официальный регламент — от одного дня до четырех недель. В среднем этот этап занимает одну неделю. Время, затраченное на этот вид деятельности, входит в общее время реализации проекта (шесть месяцев).

9.7.5. Разработка в рамках временного блока: построение

Этот вид деятельности состоит из ряда вложенных итерационных вспомогательных видов деятельности.

Построение	
— анализ	см. раздел 9.7.3 и рис. 9.15
— проектирование	см. раздел 9.7.6 и рис. 9.15
— программирование	см. раздел 9.7.7 и рис. 9.15
— тестирование	см. раздел 9.7.8 и рис. 9.15
Рецензирование пользователями	см. раздел 9.7.9 и рис. 9.15
Объединение	см. раздел 9.7.10 и рис. 9.15
Оценка возможности повторного использования	см. раздел 9.7.11 и рис. 9.15
Документирование	см. раздел 9.7.12 и рис. 9.15

Ранее для реализации на обычных языках приложений с жесткими требованиями к обработке транзакций применялась каскадная модель или V-модель процесса с централизованной структурой. Этот метод не подходит для разработки объектно-ориентированных систем. К разработке приложений такого типа лучше подходить с точки зрения итеративного метода. Цель — предоставить пользователю программу, которая будет работать правильно и просто, и достичь этого как можно быстрее. Это достигается следующим образом.

- Интенсивное повторное использование существующих компонентов, если они есть
- Применение стиля интерфейса, который совместим с другими приложениями
- Поддержка целей и задач пользователей, основанная на бизнес-процессах, а не на узко определенных функциональных обязанностях
- Оценка большого количества шагов в полной V-модели процесса со строгим регламентированием каждого из них
- Обеспечение контроля путем тестирования результатов каждого вида деятельности и установки временны*х* рамок
- Установка контроля над волновыми эффектами и неконтролируемыми итерациями — в результате этого вида деятельности появляется широко используемая программа и в качестве конечного продукта процесса, и как его составляющая
- Отсутствие требования четкого разделения между процессами производства, эволюции и сопровождения, как это было при традиционных подходах, которые обычно не обращали внимания на затраты на сопровождение при обосновании проекта

Метод временны*х* блоков сокращает время до появления продукта на рынке не с помощью каких-то волшебных фокусов, которые превращают сложные вещи в простые, а за счет создания важной, готовой к использованию части программы не больше, чем за шесть месяцев. В этом разделе подводятся итоги такого подхода.

Проект по разработке компьютерной программы делится на временны*е* блоки, чтобы повысить эффективность контроля затрат и убедиться, насколько это возможно, в реализации запланированных преимуществ, не выходя за пределы отведенного времени и средств. В конце каждого временно*г*о блока проводится сквозной контроль, рецензирование и проверка работоспособности продукта согласно составленной ранее документации. Эта оценка помогает провести как анализ недостатков, так и конфигурационный контроль, имеющий место на каждом этапе, и предоставляет информацию для принятия решения об остановке проекта, если это необходимо. Каждый временно*й* блок завершается формальной оценкой, как описано в разделе 9.7.11.

Разработчики незамедлительно выпускают первый прототип программы и могут по ходу дела в общих чертах описать разработку. Результат рассматривается пользователями и обычно представляет собой общий и поверхностный прототип. Реакция пользователей учитывается во второй итерации, когда создается более глубокий и узкий прототип. На этом этапе, вероятно, (но не обязательно) будет создан документ по разработке. Во многих случаях текст программы может создаваться непосредственно из карт классов в ОА. Когда это сделано, в результате любых изменений в разработке должны будут готовиться новые или исправленные карты классов. Это может повлиять на модель предметной области и должно быть отмечено для проверки на стадии оценки. Разработка обычно выполняется в течение 3 итераций — каждый

раз с интенсивным привлечением пользователей. Однако реальное число итераций частично определяется рамками временны*х* блоков, максимум — шесть месяцев. Нужно оставить соответствующее время для таких задач, как объединение с результатами других временны*х* блоков, интеграция с базой данных, документирование и определение классов, пригодных для повторного использования. Заметим, что разработчики не отвечают за создание классов, пригодных для повторного использования, а только за определение их потенциала. Документирование проекта до его выпуска оставляется на усмотрение руководителя, занимающегося установлением временны*х* блоков, и не обязательно производится, если это не было оговорено в начале. Важно, чтобы конечный продукт удовлетворял спецификации интерфейса и чтобы документация проектирования существовала — она не обязательно разрабатывается в том порядке, в каком производятся компоненты. Аналитические заметки на языке UML дают требуемый уровень документации и будут помещены в библиотеку для использования на других семинарах.

Во время процесса очень важно достигать поставленных целей, оценивать успехи и удовлетворять ожиданиям. Это достигается непрерывным привлечением пользователей и безупречной работой со стороны разработчиков. Другие рекомендации, о которых следует знать, можно сформулировать следующим образом.

- Четко и объективно определите временны*е* блоки и цели проекта.
- Где это возможно, оставайтесь в рамках проекта.
- Архитектура немаловажна; будьте внимательны к ней.
- Чем раньше вы рассмотрите вопросы производительности, тем лучше.
- Где это возможно, применяйте пакеты и средства высокого уровня.
- Управляйте процессом пользовательского рецензирования, чтобы не поставить слишком оптимистичных целей или, наоборот, не занизить ожидания.
- Реализуйте дисциплину неофициального контроля изменений по результатам рецензирования пользователями.
- Там, где применяется революционный подход, включите в оценки время перехода.
- Никогда не сохраняйте прототип, от которого собирались отказаться.

В рамках временно*го* блока при необходимости создается или совершенствуется объектная модель реализации (включающая объекты повторного использования, интерфейса и управления данными). Однако единственными обязательными артефактами являются работающий код, отчет по объектам-"кандидатам" для повторного использования и техническая документация. Будет ли техническая документация включать полную объектную модель реализации (OMP), зависит от привлеченных к работе руководителей проекта. Могу посоветовать создавать ее для более сложных систем. Если создается объектная модель бизнес-процесса, то OMP не обязательна. Главное — проследить, что все решения по проектированию вытекают из специфики бизнес-процесса, или непосредственно, или через OMP. Отчет по оценке временны*х* блоков должен показать эту согласованность в прототипах и/или их документации.

Временны*е* блоки охватывают виды деятельности по тестированию и должны завершаться результатами тестов, выполняемых по сценариям задач и обычному плану технического

тестирования. Тестирование должно обеспечить ответы на следующие вопросы. Сохранят ли созданные элементы свою работоспособность при усовершенствовании программы? Хорошо ли работает программа на предельных режимах или при большом объеме входной/выходной информации? Кроме того, на этом этапе создается исходный код и выполняемая программа. Руководители проекта отвечают за обеспечение того, чтобы всем элементам политики безопасности было уделено достаточно внимания и чтобы этот документ содержал контрольную таблицу и соответствующие ссылки.

Предусловия

Временной блок можно начинать, когда закончено планирование, доступны ресурсы, укомплектовано оборудование и настроена среда разработки. Кроме того, должен быть проведен семинар и разработан ОА.

Задачи

	ИТ	Пользователи	О/Р/Ф
Пересмотр целей и ОА	+	+	Р
Превращение результатов анализа в проектное решение и документирование решений по проектированию. Если необходимо, модификация карт классов. Детальная документация по проектированию разрабатывается только для сложных программ	+		О
Пересмотр существующих программ и определение существующих компонентов для возможного повторного использования	+		О
Проведение 2–5 итераций для создания прототипа (построения программы). Построение широкого и поверхностного прототипа, который постепенно превратится в глубокий и, может быть, более узкий. Сначала — основы, а затем — редкие исключения	+		О
Рецензирование каждой итерации прототипа пользователями	+	+	О
Определение возможных классов для повторного использования в будущем	+		О
Изучение вопросов безопасности, GUI, сетевых решений, кодирования и соответствия другим стандартам	+		О
Документирование	+		О
Объединение результатов других временных блоков и элементов библиотеки классов	+		Р
Проверка	+	+	Р
Документирование анализа недостатков	+		Р

	ИТ	Пользователи	О/Р/Ф
Тест по сценариям задач и стандартному техническому плану тестирования	+		О
Создание руководства пользователя		+	Р
Создание плана перехода	+		Р
Начало оценочной деятельности	+		О

Результаты и постусловия

Результат каждого временного блока — это артефакты в одной или нескольких из следующих форм.

- Программа и ее документация
- Документация любых отклонений от стратегии, т.е. системной архитектуры; стандартные компоненты
- Спецификация/отчет, содержащий анализ недостатков и другие показатели
- Служба
- Набор возможных классов для повторного использования (отчет по "кандидатам на повторное использование")

Программная система считается завершенной, когда все сценарии задач в ОА могут успешно выполняться и все тесты завершаются удовлетворительно. Вид деятельности считается завершенным, когда завершены и сама программа, и соответствующая документация, а также составлен список кандидатов для повторного использования. Программа будет пригодна к сдаче только после объединения всех необходимых составляющих. В следующих пяти разделах дается подробное описание критериев завершенности для видов деятельности, составляющих временной блок.

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть обоснование того, почему какие-либо из рекомендованных задач не включены.

Каждый временной блок может длиться от одного до шести месяцев, подчиняясь общему регламенту, и время от проведения семинаров до реализации не должно превышать шести месяцев. В идеале временной блок продолжается три или четыре месяца.

9.7.6. Проектирование

Цель этого вида деятельности — установить, что документация описывает программу достаточно подробно, чтобы ее могли поддержать те, кто не имеет непосредственного отношения к ее созданию. Он также определяет, насколько приложение удовлетворяет требованиям, изложенным в ОА.

Проектирование выполняется после уточнения результатов анализа, обычно в рамках временного блока. Во время этого вида деятельности осуществляется проектирование компьютерной программы с целью создания компьютерного решения в терминах программы, базы

данных, интерфейса и спецификаций файлов. Проектирование может выполняться до или после программирования. Однако оно предполагает, что архитектура определена заранее. И спецификация, и проектирование соответствуют микропроцессу Catalisys, о чем говорилось в главе 6.

В процессе проектирования должны соблюдаться стандарты. Язык UML — стандартная система обозначений для проектирования.

Для этого вида деятельности потребуется только ограниченное участие пользователей, хотя пользователь должен принимать участие в процессах рецензирования и тестирования на стадии завершения.

Проектирование преследует следующие цели.

- Создать техническое проектное решение, которое отвечало бы целям предприятия, одновременно учитывая производительность, требования безопасности и контроля, факторы риска, качество, гибкость, легкость повторного использования и само повторное использование.
- Определить технические детали работы системы в терминах программ, файлов и использования баз данных.
- Создать необходимую документацию для последующего сопровождения программы.
- Придать окончательный вид физическим форматам входных и выходных данных для всех видов систем. Для интерактивных систем включить структуру соответствующих транзакций.
- Оценить вероятность использования программных и аппаратных ресурсов, определяя возможные требования к изменениям.
- Пересмотреть планы разработки и реализации.

Предусловия

Завершение анализа и ОА, подтверждение результатов анализа в рамках текущей итерации временного блока.

Задачи

Задачи зависят от выбранного метода, номера итерации и от вида проекта, но в любом случае в их число входят следующие.

	ИТ	Пользователи	О/Р/Ф
Модификация модели анализа путем присваивания обязанностей и добавления объектов интерфейса	+		Р
Определение составляющих системы, интерфейсов и архитектуры	+		О
Подтверждение соответствия архитектуре и стандартным библиотекам компонентов	+		Р

	ИТ	Пользователи	О/Р/Ф
Создание подробного физического проекта и объектной модели реализации	+		Р
Рецензирование планов разработки и реализации, включая изменения в оборудовании и их влияние на сроки выполнения проекта	+		Р
Определение источника существующих данных и способа конвертирования	+		Р
Моделирование данных и разработка базы данных, схема объектно-реляционного отображения	+		Р
Проект плана тестирования программы и необходимых для этого данных	+		Р
Создание плана тестирования системы	+		Р
Создание плана тестирования приемлемости	+	+	Р
Полная документация и ее утверждение	+		Р
Сбор или создание системы показателей (метрик)	+		О
Выполнение инспектирования или неофициального сквозного контроля	+		Р
Документирование и пакетирование разработанных объектов	+		О

Результаты и постусловия

Главные результаты — это отчет по проектированию, план модульного тестирования, план системного тестирования и план тестирования приемлемости. **Отчет по разработке** (ОР) — это одна из стадий документирования проекта. Он расширяет и улучшает стадию ОА. Для этих документов рекомендуются стандартные шаблоны.

Цель **плана модульного тестирования**, написанного руководителем проекта, — подвести итоги тестирования классов и требуемых ресурсов. Его содержание должно быть примерно следующим.

1. Подход/метод тестирования, включая использование автоматических средств тестирования (Р)
2. Источник информации для тестирования (Ф)
3. Прогон теста (Ф)
4. Цель (Ф)
5. Циклы теста (Ф)
6. Условия теста (Р)
7. Используемые файлы (Р)
8. Проверенные данные, вместе с исключениями и объяснением их причин (Р)

План системного (интеграционного) тестирования резюмирует метод тестирования системы. Он сохраняется как документация к программе и учитывает внутреннюю структуру системы. Он служит для проверки соответствия работы программы ее проектному решению. Контрольные точки также нужно сверять с проектным решением. Автор плана — руководитель проекта. Вот примерное содержание плана.

1. Метод тестирования, подход, средства (О)
2. План регрессионного тестирования (Р)
3. План системной интеграции (Р)
4. План тестирования производительности (в ночное время и в интерактивном режиме) (Р)
5. Критерии завершенности (О)
6. Используемые файлы и объекты (Р)
7. Проверенные объекты (О)

Цель **плана тестирования приемлемости** — определить последовательность тестирования приемлемости системы пользователями. Он описывает метод, который используется во время тестирования приемлемости. Он сохраняется в виде документации приложения и применяется для определения, работает ли система так, как указывалось в предложении проекта, на семинарах и в отчетах по анализу программы. План тестирования дает возможность проверить, что все прецеденты могут быть выполнены так, как это было описано. Его содержание таково.

1. Метод тестирования, подход, инструментальные средства (О)
2. Критерии завершенности (О)
3. Источник информации для тестирования (Р)
4. План и график тестирования (Р)
5. Циклы теста (Р)
6. Условия тестирования (Р)
7. Определение пользователей, отвечающих за одобрение (О)

Рецензирование проектного решения подтверждает, что документация соответствует продукту. Следует проверить проектное решение, чтобы убедиться, что разработанная программа отвечает техническим требованиям и потребностям бизнес-процесса; что система безопасна, но не сверхсложна; что разработанный проект обоснован в терминах затрат и преимуществ; и что в проектировании учтены все функции, включая требования к оборудованию и программному обеспечению.

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть обосновано, почему какие-либо из рекомендованных задач не включены.

Прорецензируйте планы тестирования и отчеты, чтобы обеспечить их завершенность.

Технологии

На этапе проектирования применяется модификация процесса Catalysis, шаблоны проектирования, неформальный сквозной контроль или инспектирование для более крупных систем.

9.7.7. Программирование

Этот вид деятельности превращает спроектированную модель в работающую программу или фрагмент программы. Первый прототип обычно бывает широким и поверхностным; охватывает всю систему. Последующие прототипы будут дополнять предыдущие и рассматривать всю программу или ее часть более глубоко. Для определения компонентов, пригодных для повторного использования в последующих итерациях, используют узкие и глубокие прототипы.

При выполнении этого вида деятельности пишутся и проверяются отдельные программы. Затем эти программы объединяют и проводят тестирование их интерфейсов. Пользователи мало чем могут помочь в написании программ и связанном с этим процессе тестирования, но их помощь потребуется для определения плана тестирования приемлемости, создания примеров тестов и проверки результатов.

Информационная технология потребуется для обеспечения среды разработки, настолько приближенной к среде разработки продукции, насколько это возможно. Тест на приемлемость разрабатывается для пользователей, чтобы определить, что система работает так, как было определено изначально, и удовлетворяет требованиям бизнес-процесса.

Цели программирования таковы.

- Написать и проверить каждый программный объект.
- Убедиться, что объекты и слои связаны между собой.
- Убедиться, что система выполнена согласно спецификации и удовлетворяет требованиям бизнес-процесса.
- Установить, что все программы конвертирования работают согласно своей спецификации и находятся в состоянии, пригодном для использования.
- Убедиться, что программа работает в рамках операционных требований.
- Установить, что система в целом находится в таком состоянии, что ее можно передавать пользователям для тестирования на приемлемость.

Во время выполнения этого вида деятельности необходимо везде, где это возможно, применять специализированные средства тестирования.

Целью этого вида деятельности является проектирование, написание и проверка каждого объекта или слоя в отдельности (программное тестирование) перед тестированием системы в целом (системное тестирование).

Никакого официального одобрения не требуется.

Руководитель проекта несет ответственность за то, чтобы при выполнении проекта соблюдался план разработки.

Предусловия

Положено начало временному блоку и создана инфраструктура проекта. Спланированы рецензии пользователей. Виды деятельности по анализу и проектированию выполнены в рамках временного блока.

Задачи

	ИТ	Пользователи	О/Р/Ф
Выбрать раздел модели для написания программы	+		Р
Написать объектный код	+		О
Написать средства тестирования и модульные тесты для объектов	+		О
Объединить классы в библиотеки и описать проектные решения			Р
Создать среду тестирования системы	+		О
Проверить соответствие стандартам программирования	+		О
Проверить соответствие стандартам интерфейса	+		О
Создать среду тестирования на приемлемость	+		Р
Создать пользовательскую документацию	+		Р
Тестирование в предельных режимах	+		Р
Отметить объекты, пригодные для повторного использования	+		О
Собрать системы метрик	+		О
Начать рецензирование пользователями			

Результаты и постусловия

Главные артефакты этого вида деятельности — любые написанные программы или прототипы, пользовательская документация и техническая документация, включая метрики. Каждый объект должен сдаваться со своими средствами тестирования.

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть представлено обоснование, почему какие-либо из рекомендованных задач не включены.

Убедитесь, что продукт и документация соответствуют стандартам и что разработаны средства для тестирования.

9.7.8. Тестирование

В этом виде деятельности задействованы как разработчики, так и пользователи, и он входит во все виды деятельности, особенно в программирование и пользовательское рецензирование. Во время проектирования планы тестирования совершенствуются. Тестируемый прототип показывается пользователям и, если это возможно, испытывается ими. Разработчики должны провести модульные и интегральный (системный) тесты объектов и компонентов и убедиться, что соблюдаются все диаграммы последовательностей. В основе тестирования системы лежат прецеденты. Где это нужно, после объединения должно быть произведено массовое тестирование в предельных режимах, в идеале — с использованием программного обеспечения для автоматического тестирования.

Всюду, где это возможно, программистам следует проводить тестирование собственных программ методом "белого ящика", но не разрешайте подвергать свою продукцию тестам "черного ящика". Для больших проектов или проектов с повышенным риском рекомендуется проводить формальное тестирование внешней командой по тестированию.

Предусловия

Выпущен прототип и удовлетворены все критерии программирования. Пользователи готовы для участия в тестировании.

Задачи

	ИТ	Пользователи	О/Р/Ф
Получить планы тестирования	+	+	О
Получить средства для тестирования	+		Р
Разработать тест на работоспособность	+		Р
Провести системное тестирование прототипа, включая внешние интерфейсы	+		О
Провести тестирование на предмет соответствия прецедентам и диаграммам последовательностей	+		О
Написать отчет по программному и системному тестированию	+		О
Отослать результаты в отдел проверки качества	+		О

Результаты и постусловия

Главные результаты — это отчеты по тестированию каждого созданного объекта и системы в целом (интеграционный тест и тест приемлемости). **Отчеты по объектному и системному тестированию** подтверждают, что код объектов соответствует документации семинаров и модели, а значит готов для передачи пользователям для тестирования на приемлемость. Отчет пишет руководитель проекта. В него должны входить следующие разделы.

1. Введение
2. Заключение — подтверждение того, что система работает так, как определено в запросе на проект, на семинарах и в документации по проектированию
3. Области допустимого отклонения от спецификаций
4. Журнал регистрации ошибок и действий

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть обосновано, почему какие-либо из рекомендованных задач не включены.

Убедитесь, что результаты тестов удовлетворительны, продукция и документация соответствуют стандартам и метрики находятся в допустимых пределах.

Убедитесь, что все важные диаграммы последовательностей выполняются правильно и проверки на работоспособность прошли успешно. Убедитесь, что результаты всех тестов удовлетворительны. Результаты тестов подтверждаются подписью специалиста по тестированию.

9.7.9. Рецензирование пользователями и тестирование приемлемости

В этом виде деятельности участвуют как разработчики, так и пользователи. Тестируемый прототип передается пользователям для испытаний.

Во время этого вида деятельности пользователи, участвующие в разработке проекта, выполняют тестирование приемлемости (ТП). Тем самым устраняется необходимость общего тестирования на приемлемость, когда программа выйдет в свет. В большинстве случаев при этом все еще необходима специальная среда ТП.

Последнюю итерацию этого вида деятельности следует рассматривать как официальное тестирование приемлемости для пользователей в среде ТП.

Предусловия

Написан прототип программы и удовлетворены все критерии программирования и тестирования. Пользователи готовы к проведению тестирования.

Задачи

	ИТ	Пользователи	О/Р/Ф
Показать прототип пользователям	+	+	О
Пользователь должен испытать прототип в соответствии с диаграммами последовательностей и планом тестирования приемлемости	+	+	О
Пользователь должен испытать прототип, не следуя диаграммам последовательности	+	+	Р
Определить возможность повторного использования	+	некоторые	О
Повторить это со всеми задействованными пользователями	+	+	Р
Запротоколировать результаты проверок	+		О
Согласовать и подписать необходимые изменения	+	+	О
Согласовать следующую итерацию или завершить приложение	+	+	О
Выпустить отчет по результатам рецензирования		+	Р
Убедиться, что продукт соответствует стандартам	+		О
Начать интеграцию	+		Р

Результаты и постусловия

Единственный результат — это **отчет по результатам рецензирования**, содержащий все необходимые согласования и подписи. Он должен быть очень кратким, в идеале — меньше страницы. Этот отчет должен подтверждать, что код системы работает так, как описано в запросе на проект и документации по анализу; что руководство пользователя удовлетворительно описывает процедуры, которых следует придерживаться. Каждый член команды может внести свой вклад в этот отчет.

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть обосновано, почему какие-либо из рекомендованных задач не включены.

Убедитесь, что все важные диаграммы последовательностей выполняются правильно, проверки на работоспособность прошли успешно и все важные стандарты соблюдены.

9.7.10. Объединение, координация, повторное использование и документирование

Этот вид деятельности выполняется после ряда итераций по анализу, проектированию, программированию и рецензированию пользователями. В нем результаты данного временного блока объединяются с результатами предыдущих временных блоков или других проектов. На этом этапе утверждаются классы, претендующие на повторное использование, а также объединяется пользовательская и техническая документация. Замечания, сделанные в этом разделе, также относятся к планированию временных блоков.

Модель процесса, изображенная на рис. 9.15, предполагает, что некоторые этапы (временные блоки) могут выполняться параллельно. Понятно, что при этом возникают дополнительные сложности, связанные с распараллеливанием процесса разработки. Проекты и временные блоки необходимо скоординировать. Главная задача — убедиться, что объединение результатов временных блоков спланировано соответствующим образом и что имеет место синхронизация временных блоков. Модель, которой мы придерживаемся, показана на рис. 9.16.

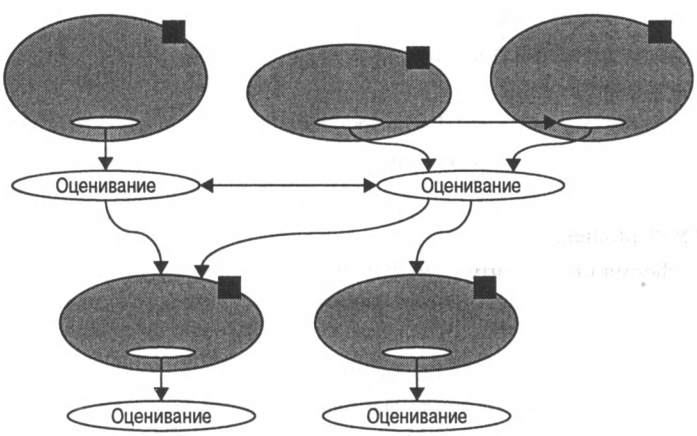


Рис. 9.16. Координация параллельного процесса разработки

На этой диаграмме показано, как в рамках каждого временного блока объединяются результаты предыдущих этапов, которые затем подвергаются оценке. В некоторых случаях временные блоки связываются непосредственно и неофициально. Иногда эти временные блоки известны только после оценки. При объединении результатов двух конечных временных блоков формируется комплексная оценка. Предусмотрительные руководители при распараллеливании проекта удваивают свои усилия по оценке результатов, а вообще, если это возможно, стараются ее избегать.

Наиболее важные вопросы, которые необходимо обсудить, касаются управления обновлением информации в архиве. Отдельные вопросы, возникающие при повторном использовании, будут рассмотрены в разделе 9.8.

Предусловия

Результаты итераций по прототипированию подписаны и готовы для отправки пользователям.

Задачи

	ИТ	Пользователи	О/Р/Ф
Определить, из каких компонентов будет состоять программа	+		О
Выполнить все задачи программирования для системы в целом	+		Р
Подтвердить информацию о кандидатах на повторное использование	+		О
Подготовить пользовательскую и техническую документацию	+		О
Прорецензировать пользовательскую документацию с участием пользователей	+	+	О
Утвердить техническую документацию	+		Р
Оценить влияние приложения на коммуникационные системы	+		О
Поддерживать связь с сетевой командой для определения требований	+		Р
Определить время отклика системы при пакетном и интерактивном выполнении	+		О
Предоставить информацию клиенту	+		О
Провести инспектирование	+	+	Р
Проанализировать недостатки	+		Р
Завершить формирование метрик	+		О

Результаты и постусловия

Система в целом готова для передачи пользователям. На этом этапе система может быть продемонстрирована в полном объеме. Следует проверить, все ли прецеденты реализованы и нет ли ошибок в каких-либо отчетах. Набор технической документации должен подтверждать следующее.

- Все объекты и решения по разработке задокументированы
- Описаны интерфейсы с другими системами

- Задокументированы любые изменения в ОА
- Составлен отчет по проектированию с применением CASE-средств
- Описаны связи между кодом объектов и их моделями, а также между моделями и спецификациями
- При необходимости включен отчет о преобразованиях
- Существует отчет о тестировании
- Задокументированы инфраструктура и требуемые действия

Команда разработчиков должна предоставить производственному отделу следующие данные.

- Все исходные коды, а также необходимые компиляторы и компоновщики
- Полные инструкции по инсталляции
- Инструкции по работе с программой

Все этапы решения задач должны быть описаны в пользовательской документации. В этот документ также могут входить копии экранов и отчеты с комментариями. В системе должны быть предусмотрены подсказки по ключевым пользовательским задачам и особенностям программы.

План реализации и перехода описывает шаги, которые следует предпринять при переходе к новой программе. Он должен содержать характерные особенности задач. План должен определять те бизнес-процессы, которые изменятся в результате реализации; должны быть определены границы изменений. При поэтапном переходе особое внимание должно уделяться тестированию. План должен включать следующие пункты.

1. Резюме
2. Подход
3. План устранения непредвиденных ситуаций, отказов и неполадок
4. Критерии завершенности
5. Необходимая подготовительная работа
6. Затраты на переход
7. Расписание
8. Границы изменений
9. Необходимые ресурсы

В отношении существующей программы этот план отражает следующие аспекты.

- Удовлетворительные критерии перехода
- Метод, которым следует пользоваться
- Работа, которую следует выполнить над существующей системой до перехода
- Схема любых отчетов, необходимых специально для перехода

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть предоставлено обоснование того, почему какие-либо из рекомендованных задач не включены. Убедитесь, что выполняются следующие условия.

- Программа работает в требуемом окружении
- Пользователи довольны результатами тестирования
- Продукт и документация завершены и отвечают стандартам
- Метрики не выходят за допустимые пределы (для регистрации ошибок накопления)
- Все важные прецеденты могут быть правильно выполнены
- Результаты проверки на работоспособность удовлетворительны
- Получены все подписи пользователей
- Документация завершена и соответствует стандартам
- План перехода удовлетворительный

9.7.11. Оценка и повторное использование

В результате оценки итогов каждого временного блока создается отчет, в который входят рекомендации по повторному использованию, заключение о качестве и краткий план реализации. Отчет должен подтверждать результаты тестирования сценариев задач и объединять все метрики этого и предыдущих временных блоков.

Оценка является частью метода проектирования, позволяющей отследить эффективность реализации каждого временного блока. Цели оценки таковы.

- Решить, стоит ли продолжать проект
- Решить, стоит ли выпускать в свет результат данного временного блока
- Определить в рамках данного временного блока потенциальные компоненты для повторного использования
- Определить, какие из классов, полученных в других проектах или временных блоках, пригодны для использования в системе
- Поэтапно перепроверить надежность и безопасность, а затем предоставить данные для качественного анализа
- Вычислить метрики проекта
- Сверить результаты выполнения с планом
- Пересмотреть и одобрить любые планы исключений
- Пересмотреть план следующего этапа
- Определить значимость существующих и возможных проблем и спланировать действия по их устранению

Повторное использование программного обеспечения повышает эффективность разработки. Разработчики отвечают за сохранность кода, так что его можно повторно использовать в последующих проектах. Главная проблема здесь в том, что для создания хороших классов, пригодных для повторного использования, нужно время и скрупулезная работа, а это противоречит принципу быстрой разработки в рамках временного блока. Перед разработчиками ставится задача рассмотрения возможности повторного использования кода для каждого отдельного проекта. В число задач оценки на последних этапах входит составление списка классов-кандидатов на повторное использование. Такие классы передаются специалисту по конструированию классов для усовершенствования, тестирования и повторного использования в последующих проектах. Разработчика классов можно оценить по частоте использования его кода в новых проектах, по количеству недостатков, упомянутых в отчетах, и по росту производительности, достигнутому в результате повторного использования.

Частью процесса оценки является анализ недостатков, который проводится в конце каждого временного блока. Количество обнаруженных недостатков сравнивается с промышленными стандартами и последними разработками, для того чтобы проследить за изменениями в лучшую сторону в вопросах производительности и точности. Для количественного измерения повышения уровня производительности и качества можно пользоваться различными метриками. В объектно-ориентированных проектах в качестве эквивалента традиционных логических бизнес-операций выступают элементарные выполняемые задачи. По возможности следует регистрировать также показатели применимости, т.е. время, затраченное на выполнение задачи, время "забывания" и т.д. Кроме всего прочего, проводится сравнение надежности (средняя наработка на отказ), затрат и эффективности программирования (количество ключевых задач, приходящихся на один человеко-месяц). Учитываются также показатели повторного использования, включая количество повторно использованных классов и новых классов, претендующих на повторное использование. Системы метрик для каждого проекта должны быть сформированы до выпуска продукта и должны учитываться на каждом этапе жизненного цикла.

Оценка — самый важный вид деятельности. Именно здесь решается, отвечают ли результаты временных блоков требованиям, следует ли продолжать реализацию, не придется ли переделывать работу или стоит отказаться от нее как от невыполнимой. В состав группы оценки должен входить спонсор и сотрудники, не являющиеся членами группы разработчиков. Во время оценки следует учитывать план обеспечения качества, в котором должно быть оговорено, какие элементы метода будут применяться и какие компоненты обязательны. Необходимо проверять соответствие проекта запланированным архитектурным и сетевым характеристикам, операциям и аппаратным средствам. При необходимости можно подключить знающих людей.

Иногда можно применять методику инспектирования и пересматривать стандарты программирования. Команда должна быть уверена, что тестирование проводится соответствующим образом, в соответствии со сценариями выполнения задач и обычными планами тестирования системы.

В оценке главных проектов должны участвовать руководители высшего звена.

Предусловия

Этот вид деятельности выполняется по окончании временного блока. Группа специалистов по оценке назначается при планировании временного блока.

Задачи

	ИТ	Пользователи	О/Р/Ф
Утвердить предложенные разработчиками кандидатуры классов для библиотеки повторного использования и предложить новые	+	+	Р
Убедиться, что тесты проводились соответствующим образом	+		О
Убедиться в достаточной производительности системы	+	+	О
Убедиться, что соблюдены соответствующие стандарты	+		Р
Выполнить тест на приемлемость	+	+	Р
Проверить удобство поддержания архитектуры	+		Р
Пересмотреть вопросы безопасности, восстановления и т.д.	+		О
Пересмотреть руководство пользователя	+		Р
Пересмотреть план перехода	+		Р
Пересмотреть план тестирования на приемлемость	+		О
Принять решение — переделывать, выпускать или прекратить разработку	+	+	О
Подтвердить конечные сроки	+	+	О
Подтвердить технические решения	+		Р
Начать реализацию	+	+	О

Результаты и постусловия

Результаты этого вида деятельности таковы.

1. Протоколы проверок.
2. Решение о том, принять или отбросить результаты этого временного блока. Заметьте, что группа специалистов по оценке не имеет права приостанавливать реализацию для устранения недостатков. Это будет сделано в ходе обычного устранения ошибок или во время следующего временного блока проекта. Они могут только посоветовать переделать работу или отказаться от нее, если проект совершенно невыполним. В противном случае их предложения составят основу нового проекта.

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть предоставлено обоснование того, почему какие-либо из рекомендованных задач не реализованы. Должны выполняться критерии завершенности предыдущих видов деятельности.

Все стороны должны подтвердить полноту и завершенность проверки. В случае дискуссии этот вопрос нужно решить простым большинством. В последнем случае свое одобрение должны дать высшее руководство и спонсор проекта. Высшее руководство должно следить за результатами рецензирования всех основных проектов.

Прорецензируйте результаты временного блока, чтобы убедиться в следующем.

- Документация разработана и план тестирования программы выполнен правильно.
- Достигнуты критерии завершения как модульного, так и программного тестирования. Программы готовы к тестированию на приемлемость.
- Программы удобно поддерживать.
- Система может быть успешно продемонстрирована в действии.
- Применяемые технологии и документация соответствуют стандартам.
- Предоставлена полная информация обо всех необходимых метриках.
- Претенденты на повторное использование определены.
- Целесообразность выполнения проекта сохраняется.
- Нет нерешенных вопросов.
- Все придерживаются политики безопасности.
- Выполняются требования законов к защите информации и другие важные законы.

Этот вид деятельности никогда не должен затягиваться дольше, чем на две недели, а обычно он продолжается всего полдня.

9.7.12. Планирование реализации

Цель этапа планирования реализации — убедиться, что все необходимые виды деятельности завершились и можно начать непосредственный выпуск программы. Этот вид деятельности состоит из трех частей: планирование реализации, обучение и инсталляция.

Если результаты временного блока успешно прошли проверку, можно приступать к реализации. В число задач реализации входят следующие.

- Обучение
- Тренировка
- Управление изменениями (через спонсора)
- Поставка программного обеспечения и аппаратных средств
- Настройка рабочего окружения и локализация
- Поддержка

Планирование реализации начинается на семинарах. Оно связано со стремлением убедиться в том, что все необходимые виды деятельности выполнены до выпуска продукта и успешного перехода к новой системе. Требования должны быть оговорены в предложениях по проекту.

Задачи этого вида деятельности состоят в следующем.

- Убедиться, что рабочее окружение настроено.
- Предоставить полный и точный план перехода и приступить к его выполнению.
- Перейти к новой системе.
- Обеспечить поддержку и помощь при переходе к новой системе и после него.
- Добиться официального принятия реализации системы.
- Убедиться, что группа управления изменениями уведомлена о предлагаемых изменениях.

Предусловия

Прежде чем завершить этот вид деятельности, необходимо получить все подписи этапа оценки, хотя большую часть планирования можно завершить и без этого. Стадия запуска проекта должна быть завершена, должны быть предоставлены результаты семинаров, быстрого анализа и главных временных блоков.

Задачи

	ИТ	Пользователи	О/Р/Ф
Завершить план реализации	+		О
Предоставить информацию о технологических требованиях группе поддержки разработки	+		О
Спланировать возможные отказы	+		Р
Выбрать и протестировать аппаратные средства и встроенное программное обеспечение	+		Р
Разработать инструкции по инсталляции и комплект программного обеспечения для установки программы	+		О
Завершить план обучения	+	+	Р
Обучить пользователей	+	+	Р
Собрать информацию о результатах тестов	+	+	Р
Подготовить план бета-тестирования	+		Ф
Провести бета-тестирование с избранными пользователями	+	+	Ф
Завершить тестирование на применимость	+	+	Р
Пересмотреть отчет по бета-тестированию	+	+	Р
Сформировать отчет об изменениях для группы управления изменениями	+		О
Получить одобрение на реализацию	+		О
Определить профили безопасности	+		Р

	ИТ	Пользователи	О/Р/Ф
Обеспечить обучение пользователей работе с новой программой	+	+	Р
Убедиться, что построение метрик завершено	+		О
Установить рабочее окружение	+		О
Выполнить переход	+		О
Протестировать результаты перехода		+	О
Одобрить переход		+	О
Обеспечить поддержку новой системы	+		О
Инсталлировать систему	+		О
Составить отчет о недостатках	+	+	О

Результаты и постусловия

Результаты этого вида деятельности включают следующее.

- Одобрение перехода пользователями, составление официального документа о том, что пользователи принимают новую систему.
- Поставка пакета реализации.

Все обязательные и рекомендованные задачи (перечисленные выше) должны быть выполнены или должно быть предоставлено обоснование того, почему какие-либо из них не реализованы.

Пересмотрите реализацию системы и убедитесь в следующем.

- Систему принимают как пользователи, так и разработчики.
- Установлено соответствующее оборудование и программное обеспечение.
- Установлена сеть (если это нужно).
- Установлены необходимые библиотеки.
- Все специальные устройства определены и доставлены.
- Вся необходимая документация системы доступна и удовлетворяет стандартам.

Может потребоваться одобрение спонсора проекта, руководителя разработки и менеджера проекта.

Этот вид деятельности входит в общий шестимесячный этап, а его реальная продолжительность должна быть меньше двух недель.

9.7.13. Планирование разработки и ресурсов

Это первый дополнительный вид деятельности, который мы рассмотрим. Планирование проекта состоит из двух видов деятельности: подробного планирования отдельного временного блока, что обсуждалось отдельно в разделе 9.7.4, и планирования всего проекта, состоящего из

нескольких временны*х* блоков. Этот последний вид деятельности неотделим от управления ресурсами для проекта в целом. Если проект состоит только из одного временно*го* блока, эти виды деятельности объединяют. В этом разделе речь пойдет о планировании разработки — вид деятельности, который начинается с глобальных и частных рассуждений и позволяет распределить работу между различными временны*ми* блоками. Параллельно с этим видом деятельности проводится поиск в архиве классов, пригодных для повторного использования. Об этом подробнее будет говориться в разделе 9.7.14. Как показано на рис. 9.15, этот вид деятельности находится на грани между проектом и организацией в целом.

Существуют планы и оценки разных уровней. На высшем уровне должен быть план, полностью охватывающий установленные границы проекта и фиксирующий задачи, касающиеся связи с другими системами, а также зависимость проекта от недостающих компонентов других систем. В этот план должен входить общий план итераций. Существует также план каждого временно*го* блока. Степень обязательности этих планов не определена. Они должны существовать, но могут быть записаны в протоколах собраний или в виде общих договоренностей между командами разработчиков. В этом смысле их не обязательно предоставлять кому-либо, кроме самих создателей.

Планирование разработки состоит из двух отдельных, но связанных между собой видов планирования: планирования самого проекта и привлечения в этот проект других наработок, а также решения вопросов взаимодействия. Еще одна проблема тесно связана с моделированием предметной области и рассматривается в следующем разделе. Руководитель разработки должен также участвовать в планировании ресурсов. Это подразумевает определение доступности рабочей силы, оборудования, программного обеспечения и других ресурсов наряду с соответствующим планированием бюджета. В этом разделе внимание фокусируется на планировании разработки приложения. Руководители проекта и разработки должны запланировать вехи для интеграции с другими проектами, службами и т.п.

Планирование проекта начинается с момента подписания предложений по проекту. Метрики должны обновляться на каждом этапе проекта. Кроме того, должен быть подготовлен план перехода, который будет использован при планировании реализации.

Руководитель проекта отвечает за начальное обучение пользователей, потому что это непосредственно влияет на успех реализации. Он определяет, кто проведет обучение. Это может быть один из пользователей, которые помогают команде разработчиков, или поставщик системы. В данном случае это должен решить руководитель проекта.

Однако пользователи также заинтересованы в том, чтобы обучение было проведено. Пользователи также отвечают за полноту требований к системе. Они должны выделить соответствующее время на обучение. Это должно произойти *до* реализации. Обучение после реализации входит в обязанности пользовательской группы.

Руководитель разработки отвечает за связь между проектом и другими проектами организации, за взаимодействие с пользователями, другими проектами, оргкомитетами и т.д.

Цели этого вида деятельности сводятся к следующему.

- Определить и согласовать ресурсы проекта с другими проектами
- Спланировать обучение пользователей
- Определить последовательность и рамки временны*х* блоков
- Определить вехи интеграции с другими проектами и потенциал для повторного использования
- Спланировать переход к новой системе

Предусловия

Предложения по проекту сформулированы и одобрены.

Задачи

	ИТ	Пользователи	О/Р/Ф
Представить проект на рассмотрение на следующем заседании оргкомитета	+	+	О
Расставить приоритеты	оргкомитет		О
Согласно выбранным приоритетам:			
Распределить ресурсы (ИТ и пользовательские)	+	+	О
Определиться с планом высшего уровня	+		О
Определить задачи временных блоков, параллельные и последовательные операции	+		О
Определить требования к обучению и получить одобрение затрат от спонсора проекта	+	+	О
Составить расписание и осуществить план обучения	+	+	Р
Завершить выполнение плана обеспечения качества (при необходимости)	+		О
Запланировать вехи для интеграции с другими проектами/инфраструктурными службами и т.д.	+		О

На начальных стадиях проекта для определения временных рамок и ключевых задач следует использовать средства оценки.

Выполнение проектов будет отражаться с помощью соответствующих средств планирования, так что руководство всегда сможет проконтролировать ход всех проектов. Напротив каждого проекта будет записано время и затраты, определенные в бюджете.

В каждой команде разработчиков должен быть человек, отвечающий за управление изменениями; за неимением такового эти обязанности выполняет руководитель проекта. В число изменений входит исправление ошибок и поправки к спецификациям. Изменения отражаются в доработанном плане, который должен быть одобрен оргкомитетом проекта.

Результаты и постусловия

Ключевые результаты этого вида деятельности — предложения для оргкомитета и план обеспечения качества. Предложения для оргкомитета составляются руководителем проекта и предоставляют достаточную информацию для определения приоритетности проекта и распределения фондов. Они содержат следующую информацию.

1. Название проекта
2. Ссылки на другие проекты
3. Краткое описание проекта и его преимуществ
4. Оценка затрат с точностью до 50%

5. Оценка ключевых задач
6. Руководитель проекта
7. Спонсор + источники необходимых материальных средств
8. Обоснование
9. Скорость выполнения
10. План проекта высшего уровня. В нем перечисляются временные блоки, указывается их последовательность и параллельно выполняемые работы
11. Бюджет
12. Требования к ресурсам, которые должны содержать требования к обучению

Качественный план помогает определить главные вехи проекта, убедиться в качестве и адекватности используемых методов (например, методов тестирования) и акцентировать внимание на необходимых изменениях. В плане обеспечения качества должны быть перечислены критерии, по которым можно судить о проекте. Он должен включать следующие данные.

1. Главные вехи (О)
2. Методы инспектирования (Р)
3. Метрики (Р)
4. Метод управления изменениями (О)

Для всех основных проектов выполнение плана обеспечения качества будет контролироваться высшим руководством. Он предназначен для того, чтобы показать ключевые компоненты, методы инспектирования, информацию для каждого вида деятельности. По завершении каждого вида деятельности ответственные должны поставить свои подписи. В число обычных методов рецензирования входят инспектирование, утверждение спонсорами и сквозной контроль.

Планирование разработки — это продолжительный, не ограниченный временными рамками вид деятельности.

9.7.14. Моделирование предметной области и управление архивом

Этот раздел касается управления архивом. Предполагается, что эту функцию выполняет группа сопровождения библиотеки.

Моделирование предметной области — это продолжительный вид деятельности по определению и усовершенствованию объектов предметной области. Он тесно связан с задачей управления архивом.

Удобно разделить анализ на различные уровни. Моделирование предметной области охватывает те объекты и структуры, которые касаются всей предметной области, а не тех ее частей, которые относятся к конкретным приложениям. Границы здесь размыты, но принцип ясен. Объекты предметной области пригодны для повторного использования, и поэтому им уделяется больше внимания при анализе, проектировании и реализации. Опытные разработчики могут их усовершенствовать, но на период совершенствования классов предметной области разработчиков нужно освободить от временных ограничений проекта. Классы,

претендующие на повторное использование, совершенствуются и проверяются в архиве для последующего применения в новых проектах.

Модель предметной области — это объектная модель, состоящая из объектов предметной области, среды работы приложений, сценариев выполнения задач и т.д. Она относится ко всему бизнес-процессу в целом и по своему общему характеру напоминает обычную модель предприятия, основанную на объектах. Вряд ли она будет хорошо работать без какого-то компьютеризированного архива, который должен хранить модели бизнес-процессов, таблицы сообщений, сценарии задач, описания классов и архитектур. Архив автоматически должен вычислять метрики и предоставлять возможность поиска как по ключевому слову, так и по иерархии.

Модель предметной области и результаты быстрого анализа обеспечивают входные данные для планирования всего проекта и отдельных временны*х* блоков. В этот вид деятельности входит принятие решения о разделении проекта на различные сегменты, которые можно реализовать независимо или почти независимо. Слои, обнаруженные при анализе, — очевидная основа для такой фрагментации. В некоторых случаях временны*е* блоки должны будут выполняться последовательно и в определенном порядке, обусловленном зависимостями. В других случаях появится возможность параллельного выполнения временны*х* блоков, если позволят ресурсы или этого потребуют сроки окончания работы. Другими словами, основная задача — сначала построить слой первостепенного значения.

Предусловия

Отсутствуют.

Задачи

	Группа программной реализации	Команда проектирования
Отвечать на запросы от команд разработчиков	+	
Составлять перечень классов для инспектирования	+	
Инспектировать классы и архитектуры	+	
Проверять архив на наличие подобных или идентичных классов	+	+
Давать информацию руководителю проекта о наличии синонимов у зарегистрированного претендента	+	
Сообщать о неполных моделях руководителю проекта	+	
Переделывать определения классов и добавлять их в архив	+	
Строить и документировать связи на уровне классов	+	+

	Группа программной реализации	Команда проектирования
Вносить изменения модели предметной области во все проекты	+	
Посещать семинары участников отдельных проектов и давать советы по поводу повторного использования архивных классов	+	
Поддерживать реестр приложений, объектов спецификации, объектов проектирования, программные объекты и их связи	+	+

Результаты и постусловия

Каждый класс, одобренный в качестве претендента на повторное использование, на стадии оценки должен быть проверен на возможность добавления его в архив. Отчет с результатами оценки будет передан руководителю проекта. В самом архиве регистрируются новые классы и связи между приложениями, спецификации, объекты проектирования и исходный код объектов. Изменения можно публиковать в ежемесячном информационном бюллетене или рассылать по электронной почте.

Связи необходимо записывать в следующей форме (как текстовые комментарии к каждому объекту и приложению).

Приложение	• список объектов спецификации в ОА
	• список объектов проектирования
	• список используемых программных объектов
Объект спецификации	• список эквивалентных объектов проектирования
	• список приложений, использующих эту спецификацию
Объект проектирования	• список программных объектов, которые реализуют этот объект проектирования
	• список связанных объектов спецификации
Программный объект	• список объектов проектирования, реализованных в коде объекта
	• приложения, которые используют этот объект

Эта задача никогда не завершается и не имеет ограничений во времени.

9.7.15. Устранение ошибок

В этом виде деятельности учитываются недостатки, неизбежные при выпуске любого нового программного обеспечения. Он включает быстрый отклик на сообщения пользователей об ошибках.

После выпуска проекта часть команды разработчиков назначается ответственной за устранение любых оставшихся недостатков. Когда ситуация стабилизируется, этот ресурс можно освободить и система переходит в полноценный режим использования.

Предусловия

Программа выпущена и ответственность за обслуживание пользователей распределена.

Задачи

	Группа разработчиков	Пользователи
Испытать систему и заполнить формы отчетов об ошибках		+
Исправить ошибки и представить регрессионные тесты	+	
Выпустить новые версии	+	
Обновить отчеты о текущем состоянии	+	+
Составить отчет о показателях недостатков	+	+
Задокументировать проблемные моменты и подать информацию в группу управления изменениями	+	

Результаты и постусловия

Отчеты об ошибках включают описание ошибки и время ее регистрации. После устранения ошибки добавляется время ее устранения.

Этот вид деятельности прекращается, когда в системе перестают находить ошибки (обычно он длится два месяца). Слишком большое количество ошибок может привести к разработке нового предложения по проекту.

9.7.16. Общие задачи и вопросы управления проектом

На рис. 9.15 жизненный цикл проекта представлен в виде диаграммы, связывающей различные виды деятельности. В этом разделе речь пойдет о задачах управления проектом и разработкой, как об отдельных видах деятельности.

Запуск проекта

При создании предложений по проекту руководитель активно привлекает к этому виду деятельности пользователей (возможно, даже создает пользовательский отдел). Затем, когда получено согласие на продолжение проекта, руководитель разработки включает этот отдел в план текущих действий.

Обычно оргкомитет проекта собирается раз в квартал. Он должен поддерживать связь с руководителем разработки, который должен (с помощью руководителя проекта) выполнять следующие задачи.

Запустить проект

- Пересмотреть предложения по проекту и убедиться, что они соответствуют стандарту.
- Добиться подписания предложений, включая оценку затрат на выполнение работы.

Подать предложения на рассмотрение оргкомитетам

- Распространить предложения среди членов соответствующих оргкомитетов.
- Убедиться, что предложение рассмотрено и ему назначен приоритет.
- Добиться одобрения предложений и перейти к семинарам, чтобы получить подтверждение в протоколах заседаний.

Спланировать первый семинар и некое подобие последовательности разработки

- Создать план высокого уровня.
- Начать процесс уяснения требований.
- Выяснить пожелания пользователей.
- Оценить затраты на разработку. Для проверки оценки затрат использовать оценочную модель.

АНАЛИЗ РИСКОВ

Перед тем как начинать любой значительный вид деятельности, следует провести анализ рисков. Для каждой системы решаются следующие вопросы.

- Что может работать неправильно?
- Какими могут быть затраты?
- Какова вероятность случайности?
- Какие меры предупреждения можно предпринять?
- Чего они будут стоить?
- Какова цена невыполнения проекта?

Необходимо рассчитать каждый фактор риска и изучить модели.

Начало работы

Начало разработки проекта связано со следующими видами деятельности.

• Одобрение	Подтверждается предварительно полученное одобрение на проведение этого вида деятельности
• Проверка плана разработки	Пересматривается последняя версия плана проекта и объясняются любые изменения

• Распределение обязанностей	Проверяется, знают ли члены проекта о своих обязанностях и временных рамках
• Управление изменениями в проекте	Согласованные изменения регистрируются в плане обеспечения качества

Выполнение проекта

Во время этих видов деятельности жизненного цикла проекта будет завершена большая часть задач, определенных в плане проекта.

Все руководители проекта должны представлять на рассмотрение оргкомитета **регулярные отчеты о ходе выполнения**, по крайней мере ежемесячно (это можно делать в устной форме). В частности, отчет о состоянии проекта должен представляться по окончании каждого вида деятельности. Руководитель проекта должен обновлять ежемесячные отчеты о ходе проекта, ориентируясь на действительность, прогнозы и другие показатели. Руководители проектов должны выполнять все установленные требования по отчетности.

Руководитель проекта должен убедиться, что выполняются процедуры **управления изменениями и контроля версий**, чтобы

- обеспечить соответствие изменений требованиям к процедуре управления ими;
- оценить влияние каждого изменения на проект.

Вехи должны отслеживаться на протяжении всего проекта. О любых существенных отклонениях от запланированных временных рамок следует докладывать руководителю разработки и спонсору.

Для проверки выходных данных каждого вида деятельности используйте процедуры **инспектирования** и другие тесты. Инспекции будут проводиться по всем главным документам и подсистемам, появляющимся в результате процесса разработки системы. В их число входит ОА, проектное решение системы, программный код и различные планы тестирования. Инспектирование обеспечивает поддержку оценок, формируемых в конце каждого вида деятельности. Каждый документ должен быть проинспектирован до того, как спонсор одобрит или подпишет его. Инспекции не могут проводиться после подписания.

Инспектирование — это формальное рецензирование документа. Его назначение — обнаружить недостатки, но не предлагать решения. Инспектирующая команда состоит из следующих членов.

- Председатель — независимое лицо, которое проверяет, все ли выполнено и все ли участники готовы и задействованы в проверке
- Читатель — тот, кому предоставляется изучаемый отчет
- Лицо, ответственное за документ
- Лицо, ответственное за входные данные для документа
- Лицо, ответственное за применение продукта, описанного в изучаемом документе

Участники группы должны знать методику инспектирования. Заседания по инспекции не должны длиться более двух часов; участники должны заранее узнать о рассматриваемых системах и продуктах, а также прочитать документ, подвергающийся инспекции. Тем не менее недостатки будут обнаружены на заседании.

По результатам инспектирования составляется список недостатков. Они исправляются человеком, ответственным за документ, под наблюдением председателя. Обычно по каждому документу проводится одно такое заседание. Однако председатель, если это необходимо, может потребовать повторной инспекции. Председатель также отвечает за сохранность контрольных листов проведенных инспекций для их использования при сборе инспекционных метрик, т.е. при анализе недостатков, обнаруженных при инспектировании.

Окончательные оценки нужны, чтобы убедить спонсора, что оценка проводится после каждого временного блока проекта. Это формальный процесс одобрения проекта перед переходом к следующему виду деятельности (или не формальный, что тоже иногда случается). В случае большого проекта в этом участвует высшее руководство. План проекта и план обеспечения качества должны быть доступны. Назначенный секретарь должен записать название проекта, стадию, которая будет оцениваться, и дату заседания. Должны быть охвачены следующие пункты.

- Статус компонентов согласно плану
- Перекрестные ссылки на виды деятельности жизненных циклов предыдущих систем
- Реальное общее количество человеко-месяцев согласно плану
- Обещанная гарантия качества
- План проверки исключений, если таковой имеется (см. ниже)
- Имя человека, ответственного за управление изменениями. Были ли внесены какие-то изменения со времени последней оценки?
- Факторы риска и зависимости (обзор на одну страницу)
- Степень готовности к следующему этапу
- Имя главного пользователя
- Результаты работы пользователей

Проект считается завершенным, когда имеется заключение по планированию его реализации и готовы компоненты или когда прекращена работа над ним. Завершение проекта — это изменение статуса проекта, сопровождаемое следующими видами деятельности.

- Завершение документации по проекту
- Заполнение формы завершения проекта

Планирование качества

Качество достигается в результате строгого соблюдения процедур тестирования, описанных в документации. Может также возникнуть необходимость в добавлении процедур, связанных со спецификой компании.

Проверка следующих аспектов	Вид деятельности
Для объектной модели	
У каждого объекта есть свой владелец	Семинары, анализ, тестирование, оценка
Каждый атрибут/метод находится на соответствующем уровне безопасности (по умолчанию, доступен только владельцу)	Семинары, анализ, тестирование, оценка
Для системы	
На каком уровне вводятся пароли: на уровне системы или сети? Если на уровне системы, то	Тестирование
должна ли система быть физически изолированной?	Проектирование, программирование, тестирование, оценка
соблюдается ли политика безопасности базы данных?	Проектирование, программирование, тестирование, оценка
соблюдается ли политика безопасности сети?	Проектирование, программирование, тестирование, оценка
с какой частотой меняются пароли?	Проектирование, программирование, тестирование, оценка
в файлы/программы не входят пароли	Проектирование, программирование, тестирование, оценка
кто устраняет старые пароли?	Планирование реализации
существуют ли способы проверки несанкционированного доступа?	Проектирование, программирование, тестирование, оценка
Общие	
Какая служба отвечает за безопасность системы?	Планирование реализации
Нужно ли создавать отдельную среду для тестирования?	Планирование реализации
Информирование отдела безопасности о выпуске/реализации системы	Планирование реализации

Вот список задач для руководителя проекта.

1. Создать предложения по проекту и добиться, чтобы его подписали.
2. Получить одобрение.
3. Определиться с командой.
4. Организовать и провести семинар по формулировке требований.
5. Получить одобрение отчета о семинарах.
6. Завершить ОА и нововведения в процесс.
7. Получить подпись под ОА.

8. Спланировать временной блок, обучение и собрать команду для оценки.
9. Начать планирование реализации, поддерживать связь с группой реализации.
10. Интегрировать и тестировать программы.
11. Выпустить отчет о недостатках.
12. Обеспечить создание и правильность всех компонентов.
13. Завершить оценку.
14. Одобрить программы.
15. Инсталлировать программы.

Маленькие проекты

Между полным циклом разработки и сокращенным, соответствующим маленьким проектам, существуют некоторые различия. Маленьким считается проект, разработка которого в целом требует не более трех человеко-месяцев, независимо от количества участников. Независимо от размера проекта, требуется следующая документация.

- Форма запуска проекта (О)
- Краткий отчет по анализу системы (включая подробности перехода) и разделы отчета по проектированию, относящиеся ко всей системе (О)
- Пользовательская документация или справка (Р)
- План тестирования системы и отчет (Р)
- План тестирования приемлемости и отчет (О)
- Документация по реализации (Р)

Следующие документы не требуются для маленьких проектов.

- Требования к обучению участников
- Документы по определению рамок проекта
- Документация по требованиям
- План перехода — включая подробный отчет по результатам анализа

Маленький проект должен пройти через те же этапы, что и любой другой, кроме семинаров. Обычно семинары официально не проводятся, но многие результаты этого вида деятельности необходимы, поэтому их получают в результате дискуссий с пользователями или опросов.

Рецензирование после реализации

После того как система просуществует на рынке порядка трех месяцев, должно проводиться рецензирование после реализации (РПР). По согласованию разработчиков и клиентов этот вид деятельности можно не проводить, если обе стороны чувствуют, что не возникнет никаких проблем и мало что придется изменять. Задача рецензирования — проверить результаты проекта с целью улучшения их в будущем, и это может иметь огромную ценность

для организации. Их также можно использовать для начала официального тестирования на работоспособность.

Если какая-то сторона требует проведения РПР, то этот этап становится обязательным для обеих сторон.

Рецензирование — это особый вид деятельности, который представляет существующую картину и изучает историю проекта для того, чтобы увидеть, какие уроки можно почерпнуть и какие изменения необходимо внести в программу или в процесс разработки. Это возможность оценить успех проекта в терминах работы системы и процесса разработки, который применялся для ее создания.

Рецензирование системы обычно проводится примерно через месяц после реализации, когда система "закрепилась". Рецензирование процесса разработки проводится сразу же после реализации. Рецензирование системы должно включать оценку ее влияния на другие системы, временные показатели и оценки сетевых потоков.

Отчет о рецензировании представляется на рассмотрение соответствующему оргкомитету.

Цели рецензирования после реализации заключаются в следующем.

- Оценить производительность системы
- Определить, отвечает ли система требованиям
- Сверить действительные затраты и преимущества с первоначальными оценками
- Пересмотреть операционные процедуры
- Оценить процедуры и методики разработки
- Оценить и измерить применимость
- Определить сильные и слабые стороны системы и решить, требуется ли более глубокое рецензирование в какой-либо области
- Оценить и систематизировать необходимые изменения
- Дать рекомендации относительно любых необходимых изменений

Результаты РПР документируются в **отчете о рецензировании системы** и **формах оценки проекта**, которые оформляет руководитель проекта. Эти документы позволяют оценить работу готовой системы и определить дальнейшие действия; они также служат для проверки процесса разработки и оценки того, можно ли его улучшить. Формы оценки проекта применяются для регистрации метрик по каждому завершенному проекту и представляют информацию для аудиторских проверок проекта. Содержание отчета о рецензировании системы должно быть следующим.

1. Резюме руководства
2. Открытия
3. Заключения и рекомендации
4. Приложение, содержащее
   термины, используемые в отчете;
   предполагаемые термины для дальнейшей работы;
   текущий анализ затрат и результатов;
   обоснования;
   метрики.

При участии пользователей РПР призвано

- подтвердить цели и рамки проекта;
- проверить операционную систему;
- проверить программу;
- проверить затраты и метрики проекта;
- завершить анализ недостатков;
- заполнить формы оценки проекта;
- подготовить документацию;
- подготовить окончательный отчет о проекте;
- представить результаты оргкомитету.

Документы и средства поддержки

Документы, связанные с процессом и поддерживающие его, должны быть примерно следующими.

- Стратегия ИТ
- Правила безопасности ИТ
- Сетевая политика безопасности
- Руководство по проектированию базы данных
- Структура приложения и стандарты компонентов
- Руководство по качеству и метрикам

Любой список одобренных средств, среди всего прочего, должен содержать следующие.

- Средства контроля и управления проектом, такие как MS Project.
- Средства документирования и автоматизации офисных операций, такие как MS Office, Visio и т.д.
- Средства для проведения семинаров и анализа, такие как Visio или CASE-средства.
- Средства проектирования, включая CASE-средства.
- Средства контроля исходного кода и управления изменениями.
- Средства оценки и составления системы метрик.
- Средства для оценки рисков.
- Языки, компоненты и библиотеки.

9.7.17. Роли и обязанности в проекте

В этом разделе рассказывается об обязанностях и ролях участников проекта. Роль семинаров была описана в главе 8.

Групповые роли

Каждый проект связан с **оргкомитетом**, который выдает разрешение на начало каждого проекта после рассмотрения предложений по проекту, а затем контролирует предварительно согласованные вехи до самого завершения проекта. Оргкомитеты работают в рамках бюджетных ограничений, определенных руководством.

Высшее руководство отвечает за обеспечение проекта необходимыми ресурсами, за выполнение определенных критериев качества, а также следит за затратами и временными рамками проекта. После определения приоритета новых проектов высшее руководство вносит их в общий список проектов.

Роли проекта

Спонсор проекта — это отдельный пользователь, который имеет право одобрять запуск проекта и его продолжение в рамках согласованных бюджетных ограничений. Он утверждает любые поправки, вносимые в реализацию проекта. Спонсор проекта несет ответственность за полную стоимость проекта и таким образом по своей значимости является ключевой особой. Этот человек должен всегда знать о всех значительных вехах и участвовать во всех ключевых проверках, особенно на стадии оценки.

Ответственность за повседневное управление проектом лежит на **руководителе проекта или разработки** как со стороны отдела ИТ, так и со стороны пользовательского отдела. Этот руководитель проекта назначает **менеджера**. Менеджер и руководитель проекта по очереди отчитываются о проекте перед оргкомитетом проекта. Руководители проекта отвечают за ежедневную связь с пользователями для обеспечения качества продукта.

Оргкомитет проверяет проекты только в исключительных случаях; он существует для принятия решений и распределения ресурсов ИТ по просьбе пользователей. Он не обязательно должен следовать предварительно согласованному расписанию и собирается не чаще одного раза в месяц для местных комитетов и одного раза в квартал для региональных, когда можно проводить переоценку приоритетов.

Отчет о оценке будет представлен оргкомитету как часть процедуры закрытия проекта.

Руководители проекта отвечают за технический и коммерческий успех проекта и за соблюдение стандартов качества во время разработки. Они иногда могут действовать как **руководители временных блоков**.

В число особых обязанностей руководителя проекта входит следующее.

- Обеспечить, чтобы вся выполненная работа соответствовала определенным задачам, содержащимся в планах проекта
- Проверить риски, связанные с программой
- Обеспечить, чтобы все используемые методы соответствовали общим целям качества проекта, затратам и временным рамкам

- Проверить качество своей собственной работы, перед тем как зафиксировать ее завершение
- Обеспечить установление всех необходимых каналов связи с другими группами
- Определить и решить вопросы и проблемы, которые влияют на качество, временные рамки и затраты на проект в целом
- Обеспечить общее качество системы в рамках процесса разработки
- Обеспечить реализацию новой системы в среде разработки
- Управлять изменениями, о которых просят пользователи
- Докладывать о состоянии дел руководству
- Диагностировать и решать проблемы

Роли разработчиков

Разработчик, конечно же, играет ключевую роль в процессе. Он может или даже должен уметь пользоваться языком Java, иметь навыки работы с операционной системой, управления финансами, обладать практическими навыками разработки, управления проектом, методами проектирования баз данных и объектно-ориентированным анализом/проектированием. В предложенном здесь методе разработчик не отвечает за создание классов, пригодных для повторного использования, а только за предложения для повторного использования. Опыт показывает, что разработчики могут выделить такие классы в ходе процесса разработки. К разработчикам не относятся аналитики, проектировщики, программисты и т.д. Речь идет именно о разработчиках. Под этим подразумевается горизонтальный уровень, а не вертикальные структуры. Стандартные ступени карьеры становятся ненужными. Это не значит, что нет разделения труда или не используются специальные навыки. Это означает плоскостную организационную структуру и гибкий подход.

Аналитикам предметной области нужен широкий диапазон знаний и умений в этой области и задатки разработчика систем; такая комбинация кажется неосуществимой. На сегодня мнение по этому поводу таково: работа должна продолжаться до тех пор, пока методом проб и ошибок не будет подобран подходящий штат. Анализ предметной области — это долгосрочная инвестиция, которая окупится, когда начнут появляться результаты семинаров. А в начале можно увидеть лишь небольшие преимущества.

Роли пользователей

Самые лучшие примеры разработки программ подчеркивают огромное значение итеративного развития, а следовательно степень привлечения пользователей должна быть достаточно высокой. Чтобы процесс был успешным, пользователи также должны предоставлять ряд ресурсов.

Пользователи отвечают за одобрение процесса на главных вехах проекта, когда спонсор проекта должен подписывать документацию. Пользователи должны проводить тестирование приемлемости, предварительно составив план тестирования. Они отвечают за поставку данных для тестирования.

Пользователи дают согласие на передачу проекта в производственную эксплуатацию и решают, был ли переход успешным. Это делается в пределах совместно согласованных временных рамок, с небольшими изменениями, одобренными оргкомитетом проекта.

Более подробно об этом говорится в разделах, описывающих каждый вид деятельности проекта в отдельности.

Пользователь не менее важен, чем разработчик. В идеале пользователь должен знать компьютер, обладать навыками управления, энтузиазмом и быть обязательным. Однако мы знаем, что свободны только неумелые пользователи, а хорошие специалисты не всегда захотят включаться в проект на полную занятость. Однако их привлечение обязательно должно быть запланировано и осуществлено.

Для решения спорных вопросов на ранних стадиях проекта должен быть назначен **ведущий пользователь**. Это должен быть человек, пользующийся уважением среди других пользователей, и он не обязательно должен посвящать проекту больше времени, чем остальные.

Руководитель проекта вполне может быть еще и разработчиком. Он должен обладать всеми ключевыми навыками управления проектом и нести ответственность за связь с оргкомитетом и спонсором.

Председатель проекта — это обычно руководитель проекта на этапе анализа. Его роль — планировать и координировать заседания по быстрому анализу и быстрому переходу и поддерживать связь с покупателями и посредниками.

Другие роли

Специалисты по оценке должны обладать знаниями о повторном использовании и техническим мастерством в данной области. Некоторые специалисты по оценке могут не входить в состав команды разработчиков и могут извлечь пользу из обучения технологиям инспектирования.

Специалист по управлению изменениями играет ключевую роль в обеспечении интеграции системы в процессе ее усовершенствования. Остальные ключевые роли принадлежат **команде поддержки** и представителям областей **безопасности системы** и **законности**.

9.8. Управление повторным использованием

Создание объектов, достаточно качественных для их повторного использования, невозможно в рамках условий быстрой разработки, ограниченной временными блоками. Автор является сторонником такого подхода, при котором для этого назначается отдельная команда повторного использования. Однако такие отдельные команды обычно имеют незначительный опыт в создании коммерческих продуктов. Причина этого — их быстрая изоляция от дел и команды разработчиков, которых они обслуживают, и превращение в "башни из слоновой кости", которые беспокоятся только о себе и которых все меньше и меньше интересуют потребности других. Чтобы преодолеть это, в [327] выдвинута идея модели "знатоков и новобранцев".

На рис. 9.17 показаны две альтернативные линии использования ресурсов программного проекта. Штриховой линией показано истинное использование ресурсов при создании проекта, а сплошная линия демонстрирует, как среднестатистический руководитель проекта распределяет ресурсы: захватывает все, что может понадобиться, вначале и удерживает их до конца временного блока. В результате этой дихотомии возникает ряд интересных последствий.

В частности, область неактивного использования ресурсов, обозначенную буквой A, можно применять для изучения и начала проекта. Область, обозначенная буквой B, определяет период напряженной работы, связанной зачастую с повышенными опасениями разработчиков. Но и область низкого использования ресурсов, обозначенная буквой C, не является идеальным вариантом для расслабления и восстановления сил; обычно это достаточно скучный период, причем всегда изматывающий. Разработчики действительно заслуживают перерыва после напряженной работы и стрессовых условий разработки, базирующейся на временных блоках. Принимая точку зрения, что изменения — это хорошее время для отдыха, можно предложить разработчикам, попавшим в одну из таких областей низкого использования, выбрать один из двух возможных путей. Им можно поручить поиск и устранение неизбежных ошибок после реализации: этот вид деятельности называют устранением ошибок. Или их можно "призвать" на субботник — поработать некоторое время с командой повторного использования или библиотеки.

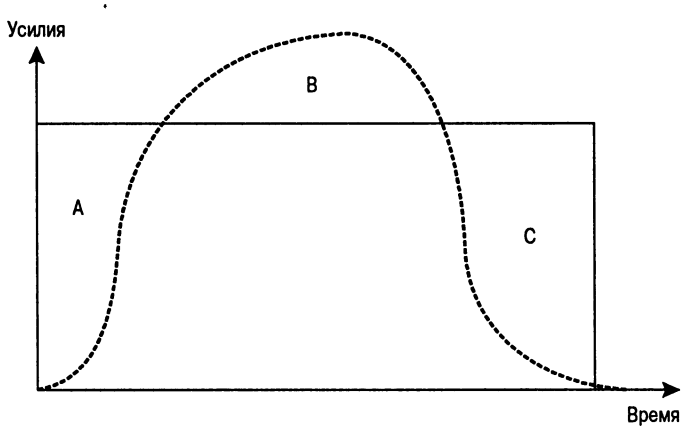


Рис. 9.17. Профили ресурсов проекта

Для выявления потенциальных претендентов на повторное использование, обобщения и создания связной библиотеки компонентов требуется особое мастерство; именно поэтому нужны специалисты по повторному использованию. В эту команду также должны входить представители группы разработчиков: люди, посвятившие себя искусству создания программного обеспечения, которые отвечают за структуру библиотеки и знают ее содержание назубок. К несчастью, таким мастерством не всегда обладают те люди, которые умеют общаться с покупателями.

Первоначально специалистов по созданию библиотеки называли "знатоками" по аналогии с системными программистами компании IBM 1960-х годов, которые часто носили бороды, сандалии и куртки с капюшоном и имели в своем распоряжении бесчисленное множество пособий для обучения людей, с головой ушедших в свое дело. Они *очень* хорошо выполняли свою работу. Полученная в результате модель управления повторным использованием, предполагающая участие в этом процессе "знатоков и новобранцев", — это модель обмена знаниями, разработанная для предупреждения ситуации, когда повторное использование становится отдельной "башней из слоновой кости". Разработчики ("новобранцы") вместе с новыми классами поставляют в библиотеку знания о ведении дела, которые они приобрели во

время разработки своих проектов, что помогает сделать эти классы более пригодными для повторного использования. Во время такого сотрудничества они перенимают у "библиотекарей" опыт повторного использования и навыки построения структур и, что самое важное, изучают, что находится в других частях библиотеки. Перейдя к следующему своему проекту, они выполняют роль послов и наставников по повторному использованию для других членов команды. Такая модель **циркуляции знаний** проиллюстрирована на рис. 9.18.

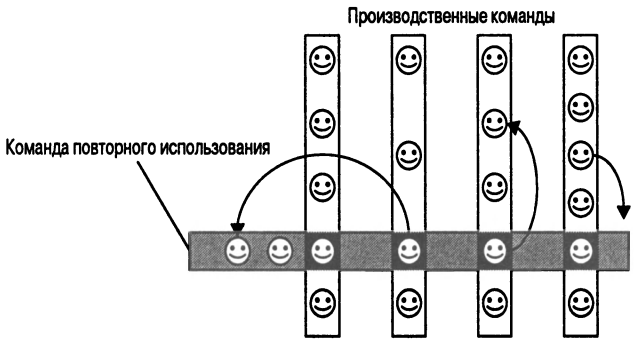


Рис. 9.18. Модель циркуляции знаний при формировании библиотеки для повторного использования

Следует отличать быструю компоновку результатов для пользователей от создания наборов компонентов (component kit), на которых строятся целые семейства продуктов. Элементы набора могут повторно использоваться много раз, и при их определении и создании необходим более высокий уровень формализации процесса проектирования; возможно, даже придется воспользоваться формальными спецификациями. Это может стать обязанностью команды, отвечающей за создание библиотеки. Опыт использования таких компонентов будет порождать новые вопросы и требования к подобным наборам и их структуре. Продукты в рамках определенного семейства можно объединить намного быстрее, и при этом не потребуется такой высокий уровень формализации. Пользователи сообщат свою реакцию разработчикам приложения и потребуют быстрой реакции на свои изменяющиеся требования. Эти два цикла обратной связи показаны в правой части рис. 9.19. Слева мы видим, как проходят процессы, в рамках которых используются более медленная и широкая обратная связь. Маркетологи пытаются понять и предугадать потребности и желания пользователей. Они разрабатывают стратегию для следующего поколения наборов компонентов. В создание новых структур и новой стратегии при развитии компонентов вносят свой вклад и разработчики.

Продукты — это быстро создаваемые специализированные решения. По возможности их нужно строить на основе актива библиотеки, пригодного для повторного использования. Такой актив должен создаваться в результате обобщения результатов проектирования уже известных продуктов и обратной связи с пользователями, а не из блестящих идей о том, что могло бы быть полезным. Команда повторного использования улучшает набор компонентов, основываясь на приобретенных знаниях, но ей следует избегать "скороспелых" обобщений: "нам может потребоваться..." и "можно было бы сделать еще и...". Они защищают свой актив и "ухаживают" за ним, предоставляя ресурсы для сопровождения своих "сокровищ проектирования". Создатели приложений выбирают и специализируют элементы из набора, а

пользователи подтверждают качество конечного продукта и его соответствие цели. Эти оценки впоследствии возвращаются к создателям набора по обратной связи.

Разработчики структур данных и стратегии их повторного использования планируют развитие библиотеки.

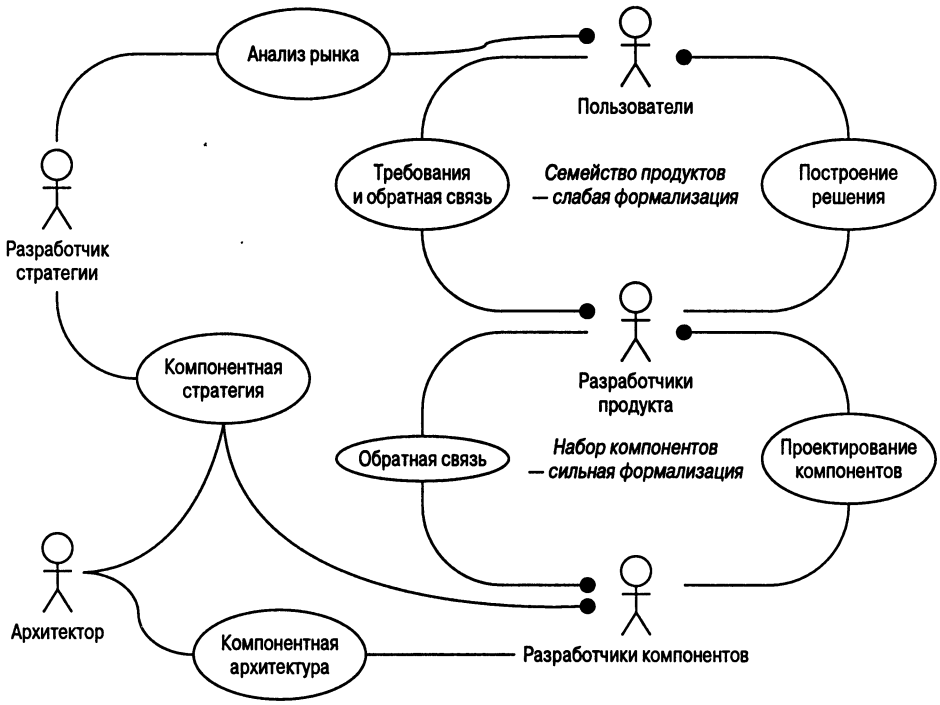


Рис. 9.19. Обратная связь при повторном использовании

Модели библиотек повторного использования

Основные вопросы повторного использования, которые мы должны обсудить, касаются координации продуктов из архива, обновления информации в архиве и управления ею. В [379] описаны четыре различные модели координации процесса повторного использования. В модели **завершения жизненного цикла** обобщение выполняется после завершения проекта. По моим наблюдениям, существует серьезная опасность того, что этот вид деятельности будет опущен, поскольку только что освободившиеся сотрудники нужны для работы с новыми проектами, тогда как ожидалось, что они займутся обобщением. Даже если это окажется возможным, как указывают Хендерсон-Селлерс и Пант, маловероятно, что покупатель будет настроен достаточно альтруистически, чтобы платить за очевидное дополнение к проекту после того, как он удовлетворительно оценил конечный продукт. Очевидный выход — обязать разработчиков создавать классы, пригодные для повторного использования, во время работы над проектом. В [543] эта модель названа G-C1. Автор данной книги называет ее моделью **постоянной разработки**. Аргументы в пользу такого подхода достаточно сильны. Затраты во время разработки можно отнести на счет покупателя. Более того, хорошие разработчики

склонны выдавать код, пригодный для повторного использования, "как шелкопряд вырабатывает шелковую нить". Его можно рассматривать как побочный продукт того, что они делают в любом случае. Однако на практике такой подход повышает затраты и увеличивает время до выхода на рынок и часто страдает от временных ограничений в рамках проекта. Очевидно, нельзя препятствовать выпуску высококачественных, пригодных для повторного использования классов во время работы над проектом, но на практике невозможно принуждать кого-либо к этому. В таком режиме мы стараемся наблюдать за поставкой и совершенствованием кода, а не за его разбиением на классы. Для преодоления трудностей, связанных с этими двумя подходами, Хендерсон-Селлерс и Пант предложили две модели, предназначенные для компаний разного масштаба: от маленьких до средних и очень больших. Это модель **двух библиотек** и **альтернативная модель затрат** (alternative cost-centre).

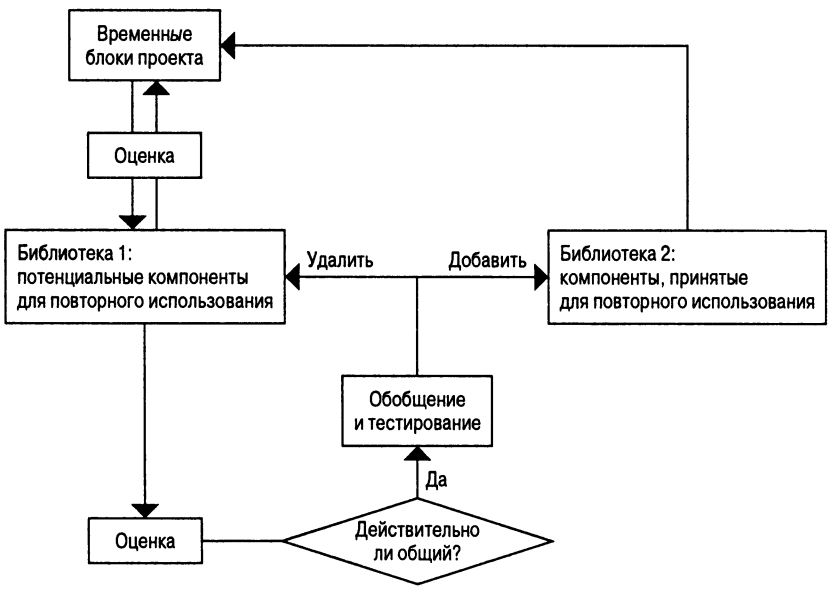


Рис. 9.20. Модель двух библиотек

Моя версия модели двух библиотек схематически изображена на рис. 9.20. Здесь показана библиотека потенциальных классов для повторного использования, определенных при работе над проектами, и еще одна, содержащая полностью обобщенные и принятые классы, обслуживаемые специальной группой. Формализовать определение пригодности объекта для повторного использования почти невозможно. Для определения результата в конкретных обстоятельствах знание дела объединяется с мастерством разработчика. Внимательный читатель уже должен был заметить, что мы приняли именно подход двух библиотек.

При этом подходе единственный дополнительный вид работы по проекту — это распознавание классов, пригодных для повторного использования; дополнительные затраты почти нулевые. Как мы уже видели, в начале проекта, на семинарах и при создании прототипов в рамках временных блоков элементы библиотеки тщательно изучаются. Если классы, выделенные таким образом, помещаются в Библиотеку 1, то на проект приходятся дополнительные расходы

по обобщению. Но именно так и должно быть, и спонсор может с удовольствием покрыть эти расходы. Модель двух библиотек непосредственно устраняет опасность излишнего обобщения, о которой упоминалось раньше.

Альтернативная модель затрат подразумевает главным образом создание и финансирование отдельного центра со своим бюджетом. Он не финансируется из бюджета проектов и изначально убыточен; затраты покрываются за счет продажи классов для использования в долгосрочных проектах. Эта модель считается подходящей только для очень больших организаций. Мой процесс тоже совместим с этим подходом, до тех пор пока применяется методика "знатоки и новобранцы".

9.9. Метрики и усовершенствование процесса

Точная оценка ресурсов и времени, необходимого для выпуска определенного проекта, — это, несомненно, самый важный вопрос для эффективного управления деятельностью компании, занимающейся развитием информационных технологий. Как правило, точную оценку требующихся для проекта ресурсов оставляют на конец жизненного цикла проекта. Если мы хотим понять и улучшить способ разработки программного обеспечения, то должны контролировать себя и свою продукцию. Другими словами, необходимо собирать показатели (метрики) как продукта, так и процесса. В число обычных метрик для программных продуктов входят количество строк исходного кода, функциональные показатели, число ошибок, обнаруженных при тестировании, и т.д. Мерой (или метрикой) процесса служит затраченное время.

9.9.1. Метрики

Исторически сложилось так, что для оценки проектов широко применялся метод SLOC (Source Lines Of Code), подразумевающий подсчет количества строк исходного кода. Но объем программы можно было узнать только после завершения работы; а до этого оценка строилась главным образом на опыте работы с подобными проектами. Затем популярность завоевал метод *анализа функций*, который предполагает оценку проекта на этапе разработки подробной функциональной спецификации. Проблемы здесь, в основном, связаны с определением критерия оценки: опыт показывает, что различные определения могут привести к совершенно разным оценкам функциональности. Кроме того, существует проблема анализа функций в чисто объектно-ориентированной среде разработки, так как в такой системе нет функций, отделенных от информационных структур, и, следовательно, нет функциональности в общепринятом смысле этого слова.

В этом разделе внимание уделяется анализу задач. Речь идет об оценке ключевых задач при определении основных требований, тех элементарных задач, которые программа помогает выполнять пользователям. Здесь есть много преимуществ. Одним из них является то, что оценка выполняется на стадии уяснения требований — достаточно рано для участия в обосновании и оценке проекта. Кроме того, ключевые задачи предметной области на стадии определения требований не предполагают формирования стратегии для их решения. Перед тем как определить подходящие показатели для метода SOMA, в том числе и ключевые задачи, мы проверили современное положение дел в области метрик объектно-ориентированных продуктов.

В методе MOSES [378] метрики формируются как часть вида деятельности по оценке качества и обеспечивают возможность тестирования кода и оценку эффективности повторного

использования. В методе MOSES различается три типа метрик: внутренние показатели классов, межклассовые внешние (т.е. интерфейсные) метрики и показатели системы. Внутриклассовые показатели связаны с размером и сложностью, межклассовые — с взаимодействиями и связью. Другие метрики связаны с познавательной сложностью. Они определяют, насколько сложно программистам понять код "чужих" классов. Метод MOSES — один из самых полных среди опубликованных методов, в которых используются метрики. Он предполагает учет следующих внутренних метрик для классов: размер, среднее число операций на класс и средний размер метода (в методе SLOC). Размер определяется следующим образом

$$W_A*A+W_M*M,$$

где A — число атрибутов, M — число операций или методов, а W_A и W_M — весовые коэффициенты, определяемые эмпирически. Обычно W_A выбирается близким к 1, а W_M находится в диапазоне от 5 до 20. Сложность измеряется по Мак-Кабу [532]. Межклассовые показатели — это средний коэффициент ветвления системы на выходе, глубина наследования и отношение числа суперклассов к общему количеству классов. Средний коэффициент ветвления не различает различные структурные отношения, в которые может вступать объект. Метод MOSES также допускает использование системы показателей, предложенных Чайдембером (Chidamber) и Кемерером (Kemerer), которые обсуждаются ниже.

Показатели MIT

Два сотрудника Массачусетсского технологического института (MIT), Чайдембер и Кемерер [161], разработали следующий набор из шести метрик системы.

Вес методов класса (ВМК) (weighted methods per class — WMC) — это сумма значений статической сложности методов класса. Если сложность любого метода принимается за единицу, это сводится к простому подсчету методов. Авторы не оговаривают, как нужно определять сложность. Вероятно, это может быть субъективная оценка или формальная мера, например цикломатическая сложность. Это внутренняя метрика для класса.

Глубина дерева наследования (ГДН) (depth of inheritance tree — DIT) — межклассовый показатель, который точнее будет назвать глубиной сети наследования. Это максимальная длина цепи специализаций или расстояние от корня дерева, когда он рассматривается как показатель класса, а не системы.

Количество потомков (КП) (number of children — NOC) — это основа классификации или количество подклассов. Конечно, это глобальный показатель системы, так как классы не знают своих потомков. Не ясно, учитывают ли этот показатель динамические схемы классификации, где КП может изменяться во время выполнения.

Связывание между объектами (СМО) (coupling between objects — CBO) — это мера связанности структур, не относящихся к одному классу. Этот критерий не делает различий между объединением, компоновкой и применением, эффективно рассматривая все это как обмен сообщениями. Это межклассовый показатель.

Отклик класса (ОК) (response for a class — RFC) — это мера структурного взаимодействия класса с другими классами и самый новый показатель из шести. Он подсчитывает количество методов, доступных классу (его собственных или получаемых за счет обращения к другому классу). Этот показатель тесно связан с коэффициентом ветвления. Согласно Чайдемберу и Кемереру, показатель ОК должен быть минимизирован, хотя это мешает выполнению субконтрактов.

Низкое зацепление методов (НЗМ) (lack of cohesion in methods — LCOM) — это также новый показатель. Он позволяет оценить неперекрывающиеся наборы переменных экземпляра, используемые методами класса. Альтернативное, операционное определение звучит так: доля методов, которые не имеют доступа к атрибутам, усредненная по всем атрибутам. Самое низкое (и самое желательное) значение НЗМ — это когда все методы используют все переменные экземпляра, а самое высокое (и нежелательное) — когда каждая переменная экземпляра используется не больше чем одним методом. Так измеряются структурные связи, но не логические или семантические, а между ними может не быть никакой корреляции. Высокое значение НЗМ может быть показателем необходимости разделения класса, но это никогда не следует делать автоматически или без должного учета семантических связей.

Эти метрики достаточно широко применялись и адаптировались при выполнении различных проектов. Например, в программном продукте McCabe Tool используются пять из этих шести показателей. К ним добавлены новые метрики и повышена степень структурирования. Это значит, что в McCabe Tool 13 показателей объектно-ориентированных систем разделены на четыре категории: метрики качества, инкапсуляции, наследования и полиморфизма. Показатели инкапсуляции включают НЗМ и метрики открытых и защищенных членов класса (Pctpub), а также метрики доступа (Pubdata). Мера наследования — это число корневых классов (Rootcnt), потомков и глубина наследования класса. Полиморфизм измеряется с помощью ВМК, ОК и процентного соотношения обращений к неперегруженным модулям (Pctcall). Качество измеряется при помощи максимальной цикломатической и внутренней сложности методов класса и числа классов, которые зависят от потомков.

Считается, что в основу шести показателей MIT были положены теоретически обоснованные положения математики и теории измерений. Используемый базис — это Вандовский (Wand) вариант математической онтологии Банга (Bunge) [773]. Это не очень убедительно, потому что сама онтология страдает от некоторых серьезных философских недостатков. Она атомистична: все понятия в ней сводятся к атомарным элементам. Более того, она вряд ли сможет противостоять феноменологической критике по поводу того, что сущность объекта не зависит от его свойств (т.е. все они могут изменяться). В [787] этот недостаток устраняется. В этой работе утверждается, что если мы воспринимаем два объекта как идентичные, то мы определили не все важные свойства, т.е. не раскрыли их сущности. Влияет ли нарушение основ на корректность суперструктуры — спорный вопрос. Если мы пришли к предложенным показателям некорректным путем, то это не обязательно должно означать, что они ошибочны.

Как отмечалось выше, по ходу работы КП может изменяться из-за динамической классификации. Я бы предложил в таких обстоятельствах фиксировать минимальное, максимальное, среднее и модальное значения КП.

Связывание зависит от взаимодействий. Может показаться, что имеет значение тип взаимодействия. Автор предпочитает собирать отдельные системы показателей для всех структур объектной модели: классификации, композиции, использования и ассоциации. Этот метод можно применять к ветвлению и ОК.

Зацепление, основанное на НЗМ, является чисто структурным. Это логическая или понятийная связь, которая важна с точки зрения повторного использования. Эти метрики должны основываться только на переменных экземпляра. Из определения не следует исключать атрибуты класса, так как они вносят существенный вклад в сложность и систему связей.

В [380] указывается, что в системах показателей MIT есть некоторые противоречия и ошибки. Авторы этой работы ссылаются на различные варианты статей Чайдембера и Кемерера [161–163], где предлагаются совершенно другие определения метрик СМО и НЗМ. В статье отмечается, что Чайдембер и Кемерер оценивают применимость метрик на основе

аксиом, предложенных в [785]. Но общеизвестно, что последние противоречивы и довольно сомнительны. Показано, что НЗМ версии 1991 года всегда равен нулю, каковы бы ни были входные данные, и что в более поздних версиях не существует различий между классами, у которых совершенно очевидны разные структурные связи. Более того, значение НЗМ увеличивается с усилением связывания, тогда как оно должно уменьшаться. Другие метрики также подверглись критике. Но на данный момент это не должно нас волновать. С нашей точки зрения, главное достижение статьи 1996 года — это новое определение НЗМ, в котором решены все эти проблемы.

Рассмотрим класс, содержащий m методов $\{M_i\}$ и a атрибутов $\{A_j\}$. Пусть $a(M_i)$ — количество атрибутов, к которым имеет доступ M_i, а $m(A_j)$ — число методов, которые имеют доступ к A_j. Тогда получаем следующее.

$$НЗМ^* = \frac{\left(\frac{1}{a}\sum_{j=1}^{a} m(A_j)\right) - m}{1-m}.$$

Это определение обеспечивает метрику, значение которой возрастает с уменьшением зацепления, нормировано и позволяет различать классы с разным зацеплением (подробнее см. [380]). Это модификация метрики Мак-Каба, описанной в предыдущем разделе, которая дает ненормированное значение, что затрудняет сравнение. Оказывается, можно построить классы, для которых НЗМ* принимает значения, выходящие за пределы единичного интервала. Однако фактически такие примеры оказываются очень плохо спроектированными с точки зрения объектно-ориентированного подхода. Поэтому автор предлагает по величине отрицательного значения НЗМ* определять степень "необъектной ориентированности" класса.

Набор метрик MIT сыграл немаловажную и продуктивную роль, а те упущения, которые были отмечены в [380], можно легко исправить, что и будет показано далее.

Сродство

Многообещающая возможность — попытка объединить связывание и зацепление, как это было сделано в метрике *сродства* [614]. Эта идея обобщает классическое замечание Константайна (Constantine) о том, что хорошее проектирование минимизирует связывание и максимизирует зацепление, определяя сродство[2] и три вида инкапсуляции. Под нулевым уровнем инкапсуляции подразумевается, что строка кода инкапсулирует некоторую абстракцию. Уровень 1 — это инкапсуляция процедур в модулях, а уровень 2 — инкапсуляция, которую обеспечивает объектно-ориентированное программирование. Два элемента системы являются сродными, если у них одинаковая история и будущее или, точнее, если изменение одного элемента обязательно приводит к изменению другого. При хорошем проектировании устраняется необязательное сродство, максимально увеличивается сродство в пределах инкапсуляции и минимизируется за этими границами. Для нулевого и первого уровней инкапсуляции это сводится к принципу связывания и зацепления. В общем, наследование уменьшает шансы повторного использования. Принцип сродства говорит, что наследование должно ограничиваться видимыми признаками или что должны быть две отдельные иерархии для наследования реализации и интерфейса. Он также не одобряет использование "друзей"

[2] Сродство буквально означает "рожденные вместе".

классов в языке C++. Пейдж-Джонс выделяет несколько видов сродства, таких как сродство по имени, типу, величине, положению, алгоритму, значению и полиморфизму. Сродство по полиморфизму особенно интересно для объектно-ориентированного проектирования и тесно связано с проблемами немонотонной логики. Например, если *летать* — это операция класса *птиц*, а *пингвины* — это подкласс *птиц*, то операция *летать* в одних случаях может выполняться, а в других — нет. Если система будет меняться, здесь могут возникнуть проблемы в сопровождении. Чтобы избежать таких проблем, можно применить правила (как в методе SOMA), а также использовать нечеткие объекты и наследование. Сейчас еще не совсем ясно, как сродство можно измерить на практике.

Еще одну попытку объединить связывание и зацепление можно найти в [198], где для иллюстрации сосуществования объектов различных уровней внутри одного приложения используется аналогия с аппаратными средствами.

В методе SOMA значения атрибутов могут быть нечеткими множествами, а к связям наследования может быть добавлен коэффициент достоверности. Неизвестно, может ли эта нечеткость значений атрибутов или классификации уменьшать сложность, определяемую обсуждаемыми выше метриками. Интуитивно можно ожидать, что введение нечеткости снизит сложность проекта, аналогично другим областям применения этой методологии. В частности, при нечетком управлении необходимо меньше правил, чем при использовании традиционных подходов. Такой же вопрос возникает и для множественного наследования. Возможно, применение множественного наследования снизит BMK, но повысит значения некоторых межклассовых показателей.

Еще один открытый вопрос — как учесть тот факт, что целое может быть больше (более связным), чем сумма его частей.

Применяемые метрики часто зависят от оцениваемого материала. Те, которыми пользуются разработчики классов предметной области, могут не подходить разработчикам приложений и наоборот.

Другие подходы

В [496] описан опыт авторов по управлению объектно-ориентированными проектами, предложен список показателей, подходящих для таких проектов, а также даны полезные эвристические рекомендации по их применению. Некоторые из этих метрик общеизвестны (такие, как количество методов на класс), а некоторые являются нововведениями (такие, как количество вспомогательных классов на ключевой класс и количество сценариев). Вспомогательными называются классы, которые обеспечивают решение задачи, но не играют существенной роли для моделирования предметной области. Автор предлагает назвать их "классами реализации", например это такие классы, как `Stacks`, `Windows`, `Buttons` и т.д. Используемые сценарии имеют тот же недостаток, что и прецеденты: нельзя выделить их "стереотипный" характер или свести к элементарным составляющим. При описании метода SOMA показывается, что такое разложение может привести к новому показателю.

Предлагаемые советы достаточно разумны и основаны на опыте авторов. Однако, с теоретической точки зрения, это всего лишь относительный список показателей, не имеющий никакой связи с теорией и неопровержимых доказательств для включения в него или, наоборот, исключения из него каких-либо отдельных метрик. Мало внимания было уделено другим работам в этом направлении (особенно работам Хендерсона-Селлерса из Австралии и его учеников). В книге критикуется всего один источник — [161] — за то, что он "основывается на теории". Автор считает это оскорбительным: разумный подход к созданию систем метрик

должен основываться и на теории и на практике. Такой фундамент, возможно, сократит неимоверно длинный список показателей, приведенный Лоренцом, и смягчит относительный характер некоторых из его показателей. С другой стороны, Лоренц дает ценные эвристические советы относительно пороговых значений, которые нельзя найти ни в какой другой литературе.

В [151] представлены три набора метрик для анализа, проектирования и реализации, каждый из которых связан с абсолютно неформальной моделью микропроцесса. Предложенная идея анализа сильно отличается от идеи проектирования требований, развиваемой автором данной книги. В [151] анализ рассматривается как процесс переработки существующего документа по требованиям. Кроме того, в этой работе намного сильнее, чем в данной книге, подчеркивается центральная роль диаграмм переходов между состояниями. С другой стороны, в [151] также используются прецеденты и важное внимание уделяется определению онтологии предметной области или словаря, аналогично тому, как это делается в методе SOMA. Процесс анализа состоит в определении вклада каждого компонента или артефакта в производство других артефактов; особенно в построение диаграмм классов и диаграмм последовательностей. Метрики анализа, в основном, базируются на подсчете таких показателей, как число записей словаря, количество вхождений прецедента и класса в различные диаграммы. Они применяются для прогнозирования действий, связанных с производством артефактов анализа. Опасный момент здесь — это, конечно же, фиксация результатов анализа, которая присуща структурным методам; внимание должно фокусироваться на конечном продукте, а не на промежуточных компонентах. Но в бюрократичных компаниях такой подход мог бы стать популярным. В предлагаемый набор вошли все показатели Чайдембера, кроме НЗМ.

В [151] огромное значение придается тому, что аналитические объекты должны рассматриваться как независимые, параллельные процессы с собственными потоками управления, несмотря на то что этот скрытый параллелизм должен устраняться при проектировании, потому что объекты реализации обычно пользуются одним и тем же потоком в рамках процесса. Много внимания уделяется устранению ассоциаций высокой "арности". В методе SOMA этот механизм не нужен из-за настойчивого требования использования однонаправленных отображений вместо двунаправленных ассоциаций. Процесс проектирования сводится к созданию модели выполнения, главным образом, на основе диаграмм переходов из одного состояния в другое. Затем создаются кластеры для каждого распределенного процесса, строится их последовательность и выполняется оптимизация. Метрики проектирования связаны с оценкой качества и объема работ. И наконец, все представленные показатели реализации относятся к языку C++. Однако метрики проектирования и реализации — это существенное дополнение к показателям анализа. Показатели продукта имеют много общего с набором метрик бизнес-объектов в методе SOMA, который будет обсуждаться в следующем разделе.

Еще в одной интересной книге о метриках объектно-ориентированной разработки [787] проблема метрик также связывается с процессом разработки. Подход Уитмайра (Whitmire) — это, несомненно, самый теоретически обоснованный подход из тех, которые обсуждались до сих пор. Больше всего заслуживает похвалы его попытка предоставить прочное основание как для своих показателей, так и для моделирования объектов вообще. Он делает это при помощи теории категоризации — отрасль прикладной алгебры, которая изучает абстрактные математические структуры, и теории измерений. Короче говоря, он рассматривает класс как категорию, *объектами* которой выступают атрибуты предметной области и пространства состояний, определяемые с их помощью, а *связями* — теоретико-множественные проекции картезианского произведения этих предметных областей и функций, представляющих переходы из одного состояния в другое. Другими словами, класс — это *диаграмма* связей. Потом

эти категории становятся объектами в спроектированной модели класса. Связи являются функциональными элементами и представляют собой ассоциации наследования, агрегации и отношения использования. Для нетривиальных классов эти диаграммы становятся чрезвычайно сложными, и поэтому данный метод может оказаться слишком сложным для средней организации-разработчика, но это действительно огромный шаг вперед в нашем понимании и показателей, и основ моделирования объектов. В подходе Уитмайра есть одна небольшая проблема. Не совсем ясно, могут ли классы в этой модели быть объектами. Теорию можно было бы дополнить таким образом, чтобы она была совместима с методом SOMA. Множественное наследование также не принято во внимание. Некоторые специалисты могут счесть эти категории достаточно патологичными, так как в данной теории отсутствуют связи, представляющие ассоциации (кроме наследования). Они совсем не похожи на категории, которые используются в традиционной математике, и подразумевают возможность различных формулировок. Уитмайр дает рецепт разработки метрик, а не просто предлагает список того, что можно измерить, так как это делается во всех остальных подходах, обсуждавшихся в этой главе. Это опять же делает его подход достаточно сложным для обычных организаций. Главное отличие состоит в том, что он предлагает способ измерения качества проектирования. Для всех разработанных метрик устанавливаются формальные свойства, и значительное внимание уделяется правильному составлению шкалы измерений.

Метрики метода SOMA

Набор показателей, предлагаемый в рамках метода SOMA, представляет собой попытку синтезировать и пополнить методики MOSES, MIT и принципы Лоренца, описанные выше. Следует также отметить, что методы MOSES и SOMA сводятся к единому набору метрик в рамках процесса OPEN.

Таких показателей, как BMK, недостаточно для описания сложности объекта метода SOMA, так как этот показатель отражает сложность, обеспечиваемую в результате применения наборов правил. Наши показатели учитывают это, но не позволяют измерить влияние формальных утверждений, в которых они используются. Этот вопрос открыт для обсуждения, так как, безусловно, некоторые утверждения заменяют правила, и это следует учитывать. С другой стороны, это намного сложнее автоматизировать. Автор предлагает учитывать следующие показатели. В любом случае при помощи соответствующего программного обеспечения метрики можно формировать автоматически на уровнях требований, спецификации и кодирования.

Приведем метрики для объектной модели предметной области.

BM1. Взвешенная сложность WC_C класса C определяется как

$$WC_C = W_A*A+W_M*L_M*M+W_R*N_R*R,$$

где

- A — количество атрибутов и ассоциаций;
- M — количество операций/методов;
- R — количество наборов правил;
- N_R — количество правил в наборе, умноженное на среднее число антецедентных выражений для правила;

- L_M — это пропорциональное превышение числа строк кода в методе над согласованным стандартом, зависящим от языка (этот коэффициент обеспечивает корректность уравнения для любой размерности);
- W_A, W_M и W_R — весовые коэффициенты, полученные эмпирически.

BM2. Коэффициенты ветвления для всех четырех структур вместе со своими средними значениями.

BM3. Глубина двух ациклических структур: `Dclass` и `Dcomp`. Этот показатель является обобщением глубины дерева наследования.

BM4. Количество абстрактных и конкретных классов.

BM5. Количество объектов интерфейса, предметной области и приложения, которые входят в проект.

Следует заметить, что здесь собрано достаточно информации для восстановления всех показателей MIT, кроме НЗМ.

Приведем метрики для модели действий.

AM1. Взвешенная сложность каждой задачи T

$$WC_T = W_I*I+W_A*A+W_E*E+W_R*N_R*R,$$

где

- I — количество объектов (в грамматическом смысле) на одну задачу. Обычно этот показатель соответствует количеству именных конструкций;
- A — количество связанных задач (если они есть);
- E — количество исключений или альтернативных сценариев;
- R — количество наборов правил;
- N_R — количество правил в одном наборе, умноженное на среднее число антецедентных выражений в одном правиле;
- W_I, W_A, W_E W_R — определяемые эмпирически весовые коэффициенты, которые могут быть равны нулю, если экспериментально доказано, что такой фактор, как E, не имеет никакого влияния.

AM2. Коэффициенты ветвления для всех четырех структур в модели прецедентов вместе со своими средними значениями. Например, количество исключений (альтернативных сценариев) на одну задачу (коэффициент ветвления использования).

AM3. Глубина двух ациклических структур: `Dsub` и `Dcomp`.

AM4. Количество абстрактных и конкретных классов.

AM5. Количество объектов интерфейса, предметной области и приложения, которые входят в проект.

AM6. Количество внешних агентов, сообщений и внутренних агентов в объектной модели агентов.

AM7. Количество элементарных задач, которые не имеют подзадач: узлы-листья дерева задач.

Последний показатель, который называют количеством **ключевых задач**, — самый важный и самый новый среди показателей метода SOMA. Это потенциальная замена функций как меры общей сложности, обеспечивающая дополнительное преимущество автоматического

сбора информации. Более того, его можно оценить на более раннем этапе жизненного цикла — во время установления требований.

Число строк кода SLOC — плохое средство оценки, потому что оно сильно зависит от языка, окружения и программиста, а оценка нужна еще до того, как будет принято какое-либо решение о языке написания системы. Функциональные показатели, позволяющие решить эту проблему, не подходят, поскольку их получение сопряжено с большими усилиями и они плохо отражают специфику объектно-ориентированных систем, приложений, управляемых событиями, программ реального времени, систем с интенсивными вычислениями и т.п. Нужен показатель, не зависящий от кода, который можно собирать на стадии установки требований и после нее. К счастью, объектная модель задач метода SOMA предлагает как раз такой показатель. Этот показатель обладает дополнительным преимуществом, которое заключается в том, что его можно собирать автоматически. Эта метрика — количество ключевых задач.

Ключевые задачи представляют собой элементарные задачи, которые система помогает выполнять пользователю. В их число входят автономные задачи, которые выполняются внутри системы при решении видимых извне задач. Задача является **элементарной** по отношению к поставленной проблеме, если ее дальнейшее разложение приведет к понятиям, не относящимся к предметной области. Например, задачу регистрации можно разложить на подзадачи, такие как "послать подтверждение" или "проверить данные" и т.д. Мы можем продолжать разложение до того момента, пока это не приведет к предложениям типа "нажмите клавишу". Такие понятия уже не относятся к предметной области, хотя и могут входить в область разработки пользовательского интерфейса. Это четкое правило, хотя оно и основано на нечеткой оценке того, относится понятие к предметной области или нет. На практике почти всегда видно, когда следует прекращать разложение. Элементарные задачи — это конечные узлы (пункты) дерева задач (сети), и их подсчет можно автоматизировать, считая задачи, не имеющие составляющих подзадач.

Предполагается, что каждая задача выражается предложением, состоящим из подлежащего, сказуемого, прямого дополнения, предлога и непрямого дополнения. Этот подход также подразумевает некоторое мастерство и согласованность разработчиков в определении уровня элементарных задач. Рекомендуется, чтобы один человек (в идеале — ответственный за библиотеку классов) проверял все модели задач. В конце концов следует установить какие-то правила и придерживаться их.

Метрика ключевых задач имеет значительные преимущества перед оценкой функциональности: ее можно накапливать автоматически и в более ранний период жизненного цикла. Более того, эти показатели разработаны специально для объектно-ориентированного подхода к построению систем.

Модели оценки

Оценка — это ключевая задача в разработке программного обеспечения. Большинство традиционных методов оценки полагается как на опыт работы с такими проектами и возможность их сравнения с подобными проектами, так и на показатели, которые зависят от количества строк кода, который должен быть создан. Ни один из этих методов не подходит для оценки объектно-ориентированных систем. Эвристические методы не будут работать просто из-за того, что в мире еще недостаточно опыта работы с такими проектами. Некоторые специалисты заявляют, что подобные показатели приводят к необъективности даже в обычных проектах. Кроме того, количество строк исходного кода также определяется опытом разработчика.

Этот показатель можно использовать только для таких языков, как COBOL, на котором были написаны многие приложения в рамках одной и той же организации. Очевидной единицей оценки для объектно-ориентированного программирования или проектирования является объект. Конкретная методика с соответствующими показателями для объектно-ориентированной оценки была предложена в [470]. Единственная из традиционных методик оценки, которую можно хоть как-то применить, — это анализ функциональности [18]. Этот метод может дать грубую первую оценку, если подходить к нему осторожно и критически.

Для оценки трудозатрат предлагается использовать ту же модель, на которой построены почти все остальные оценочные методики.

$$E = a + pT^k,$$

где E — трудозатраты в человеко-часах, T — количество ключевых задач, p — величина, обратная производительности (которая определяется эмпирически), k и a — постоянные. Величину a можно рассматривать как затраты на запуск проекта и накладные расходы. Производительность сама по себе — это функция уровня повторного использования. Она может зависеть от отношения объектов предметной области к объектам приложения в *объектной модели предметной области* (ОМПО) и от сложности ОМПО, основанной на взвешенной сложности класса, коэффициентах ветвления и т.д. В этой области еще предстоит сделать много экспериментальной работы.

9.9.2. Усовершенствование процесса

Сбор показателей имеет первостепенное значение для усовершенствования процесса, потому что невозможно объективно оценить степень оптимальности процесса, пока нет показателей качества как процесса, так и самого продукта. Все процессы слишком похожи друг на друга [572]. Следовательно, кроме метрик метода SOMA, нужно измерить недостатки системы, время наработки на отказ, стоимость возмещения ущерба и другие показатели продукта, описанные выше. Кроме того, нужно измерить производительность: время, затраченное на проект, использованные ресурсы и т.д. В число показателей также следует включить косвенные факторы: уровень образования разработчиков, рабочую обстановку и даже личные качества. Затем можно начинать искать пути усовершенствования и выполнять требуемую оценку. Очевидный успех — это хорошая мотивация разработчиков и завоевание доверия спонсоров.

Сейчас модно строить программы по усовершенствованию процесса в рамках так называемой "модели зрелости процесса", примерами которой являются модели, предложенные в [400] и в институте программной инженерии SEI (Software Engineering Institute). Предложенная SEI модель зрелости возможностей завоевала неоправданное доверие, потому что правительство Соединенных Штатов Америки не выкупает организации, не отнесенные к некоторому уровню "зрелости" в рамках этой модели. Согласно этой модели, организации делятся на пять относительных категорий (Initial, Repeatable, Defined, Managed, Optimizing), которые расположены в порядке возрастания. Абсурдность этих названий абсолютно очевидна! Но это еще не самое худшее. Оказывается, затраты, необходимые для достижения высшего уровня, почти астрономичны — и вам придется заплатить за такую аккредитацию. Насколько я знаю, только одна организация в мире находится на этом уровне. Очень хорошо об этом сказано в [425]: "Некоторая неоднозначность при отражении сложности реального мира в рамках ограничений искусственной систематики приводит к возникновению "касты"

специалистов-консультантов, которые помогают разбираться в этих таинственных вопросах". По этим причинам я против подобных моделей зрелости процесса.

9.10. Проектирование пользовательского интерфейса

В рамках процесса разработки отдельно выделяется подход к проектированию пользовательского интерфейса. В этом разделе мы остановимся на некоторых его принципах.

История пользовательского интерфейса

Пользовательские интерфейсы появились вместе с самыми первыми компьютерами. Тогда они представляли собой основанный на электронно-лучевой трубке дисплей, на который можно было выводить арифметические символы, и штекерную панель для ввода. Программирование тогда напоминало перемонтировку оборудования и расшифровку полученных волшебных знаков. Кроме того, первые машины издавали достаточно громкие звуки, так что опытные операторы могли догадаться, что именно они сейчас выполняют, и обнаружить отклонения от нормы, такие как бесконечные циклы (не прекращающееся жужжание). Позднее для ввода информации начали использовать перфокарты Холлерита, а для вывода — принтеры. Это был значительный шаг вперед в смысле удобства для пользователя — пока он не уронит большую коробку перфокарт. Главный недостаток этого подхода дает о себе знать, когда программист пропускает запятую в строке программы и через день или два обнаруживает, что программа не компилируется всего лишь из-за этой маленькой ошибки. Бумажная лента не боится падения, но легко рвется. Время обратной связи значительно сократилось с появлением возможности дистанционного ввода заданий. Программисты получили возможность передавать свои задания по телетайпу, который был связан с удаленным компьютером при помощи модема. Когда телевизионные технологии стали более дешевыми, эти терминалы были постепенно вытеснены экранными терминалами, которые имитировали работу устройств диалогового дистанционного ввода заданий. Именно по этой причине эти дисплейные терминалы были названы "стеклянным телетайпом". В 1970-х годах дисплеи начали применять не только программисты, но и обычные пользователи. Интерфейс командной строки дал возможность упростить систему меню и ввод данных. Примерно в это же время Дуг Энгельбарт (Doug Engelbart) изобрел мышь, но тогда еще никто не пользовался ее возможностями.

Впервые мышь нашла свое широкое коммерческое применение в процессе работы компании Xerox PARC над интерфейсом Star и позже была реализована в интерфейсах Apple Lisa и Macintosh. Грызуны[3] обеспечивают совершенно другой способ взаимодействия пользователя с дисплеем, который основывается на модельном представлении действий указания и выбора, а не на инструкциях и описаниях. Стиль компании Xerox/Apple был скопирован многими другими и в конце концов превратился в знакомые нам сегодня стандартные системы, такие как MS Windows и X-Windows. Были попытки выйти за пределы интерфейса в стиле Windows, некоторые из них основывались на технологиях виртуальной реальности, но достичь согласия по поводу будущего стандарта пока не удалось.

[3] Можно называть их именно так, потому что были разработаны педальные мыши, которые бегают по полу. Их и больших мышей, которые используются специалистами по графике, иногда называют **крысами**.

Почему именно GUI

Графический пользовательский интерфейс (Graphic User Interface — GUI) популярен благодаря следующим качествам. Он прост в использовании, способствует повышению производительности пользователя и разработчика, обеспечивает более высокий уровень комфорта и снижает вероятность стресса. Сам по себе GUI не обеспечивает повышения производительности; для этого требуется хорошо разработанный GUI. Использование интерфейса, как мы увидим позже, зависит и от типа поставленной задачи. Как по мне, самые большие преимущества можно получить при постоянстве интерфейса. В Windows, например, независимо от того, какое приложение выполняется, меню File всегда находится в левом верхнем углу экрана, а Help — в правом. Даже если у вас нет мыши, в MS Windows *всегда* можно закрыть приложение с помощью комбинации клавиш <Alt+F+X>. Исследование приложения — это изучение пользователем его структуры без совершения каких-либо действий. Возможность исследования часто достигается "затенением" невыполнимых в текущем режиме позиций меню; например, нельзя *вырезать* что-то, пока вы его не выделили, но при этом вы можете видеть, что система поддерживает команды *вырезать* и *вставить*. Функция *отмена последнего действия* также создает благоприятные условия для исследования, так как можно попробовать выполнить какое-то действие, а затем отказаться от него, если эффект окажется нежелательным. Еще одно преимущество — это модельное представление так называемого рабочего стола, которое помогает пользователям переносить свои рабочие привычки при работе с компьютером. Ниже мы подробнее обсудим влияние такого переноса.

Существует множество ситуаций, вызывающих необходимость применения GUI. Его можно использовать, чтобы скрыть многоязыковую реализацию (части системы могут быть созданы с помощью различных продуктов). Самое распространенное назначение GUI — облегчить пользователю способ взаимодействия с машиной или оградить от сложностей сети. Если GUI строится при помощи объектно-ориентированного языка, то его можно рассматривать как пример объектной оболочки.

Несомненно, главные преимущества графического пользовательского интерфейса — это согласованность, простота в использовании и изучении, а также легкость взаимодействия различных приложений. Однако существует ряд препятствий, среди них — дополнительное программное обеспечение, оборудование, затраты на выполнение и сопровождение и более высокая сложность программного обеспечения. Тот факт, что размер памяти, необходимый для ПК начального уровня, вырос за два года больше чем в четыре раза, а размер жесткого диска увеличился почти на порядок, — это дань потребностям GUI. С другой стороны, цена такой машины за этот же период изменилась совсем незначительно. Что бы вы ни думали по этому поводу, несомненно то, что большинство организаций будет устанавливать GUI и ориентироваться на них. Эти компании неизбежно будут пользоваться объектно-ориентированными средствами, и им будут нужны соответствующие методы разработки программного обеспечения. Все эти дополнительные затраты окупятся лучшими, более удобными в применении, более гибкими приложениями и вытекающей отсюда прибылью.

9.10.1. Проектирование человеко-машинного взаимодействия

Проектирование человеко-машинного взаимодействия (ЧМВ) напоминает другие задачи проектирования, и здесь работают те же принципы. Спроектированные артефакты должны соответствовать своему назначению. Их поведение должно быть естественным и отвечать ожиданиям пользователей. Не должно быть никаких неприятных сюрпризов, кроме тех,

которые вводятся сознательно в качестве аварийных сигналов. Использование артефактов должно сказываться на выполнении поставленной задачи. Они должны соответствовать умственным и физическим способностям пользователей. Самый распространенный пример плохой разработки интерфейса, который не связан с компьютерами, — это дверные ручки. Недавно мы с коллегой зашли в офис, и когда на нашем пути встретилась закрытая дверь, я взялся за ручку и толкнул дверь. Ничего не произошло, потому что дверь открывалась только на себя. Мой коллега рассмеялся и заметил, что его старый парикмахер закричал бы: "Глупый мальчишка! Неужели ты не видишь, что ручка здесь для того, чтобы тянуть за нее?" Мне пришлось объяснять, что больше двух лет я проработал в здании, где у всех дверей ручки были с двух сторон, и мне часто приходилось напрягаться, пытаясь открыть на себя двери, которые следовало толкать. В конце концов я полностью потерял ориентацию в своих отношениях с любыми дверями. Я до сих пор считаю просто невыносимой глупость строителей и архитекторов. Более того, за такую работу им еще и платят, что еще более глупо! Мне известны, по крайней мере, две компьютерные системы, которые выдают абсурдный совет: "Для выхода из системы нажмите клавишу <Enter>[4]". Автору пришлось долго и терпеливо объяснять пользователям компании Mac, что для закрытия Windows нужно щелкнуть на кнопке **Start**. Это же очевидно! Разработчики будут продолжать выпускать такие глупые интерфейсы до тех пор, пока пользователи будут продолжать их покупать.

Можно найти множество примеров такого глупого проектирования. Мне кажется, что хорошая разработка — это та, которая поддерживает диалоговый стиль взаимодействия. В [54] компьютер сравнивается со слугой и высмеиваются разработчики плохих интерфейсов. Как бы отреагировали разработчики, если бы слуга отвечал на их простые вопросы предложениями типа "Ошибка программы 42, сэр!"? Еще хуже, когда компьютер ведет себя как слуга, которого попросили положить новую рукопись писателя-фантаста в шкаф для хранения документов, а он возвращается с комментарием: "Извините, сэр, но шкаф был полон, поэтому я сжег рукопись" [54]. Нечто подобное недавно произошло со мной.

Критиковать чужие разработки намного легче, чем самому придумать что-то хорошее. В этом длинном разделе я попытаюсь дать ряд рекомендаций для хорошей разработки пользовательского интерфейса, включая хорошие практические советы по анализу. Ключевая методика — это анализ задач, который широко используют разработчики успешных продуктов. Главный вывод, который можно сделать, — это то, что для создания удобных, полезных и правильных программ очень важна ориентированность разработки на пользователя. В первую очередь рассмотрим вопросы аппаратных средств.

Выбор аппаратных средств

Аппаратные средства для пользовательского интерфейса удобно разделить на средства ввода, вывода и того и другого вместе. Существует много устройств ввода, в число которых входят штекерные панели, перфокарты, бумажные ленты, клавиатура (стандартная и другая), мыши, шариковые манипуляторы, лучевые ручки джойстика, графические планшеты, сенсорные экраны, микрофоны и т.д. Существует несколько видов сенсорных экранов, основанных на инфракрасном сканировании, свойствах акустической волны, электропроводящей мембраны и сопротивления. Мы не будем обсуждать эти вопросы подробно, но, вероятно, все согласятся, что сенсорные экраны подходят для большинства приложений. По мнению многих, *голосовой ввод* (voice input) был бы идеальной формой ввода информации для большинства

[4] Enter — по-английски "войти". — *Примеч. ред.*

задач. Здесь не нужно тратить время на изучение языка, потому что мы уже владеем своим родным языком. При этом наши руки и глаза свободны, а нагрузка значительно снижается благодаря общему характеру речи. С другой стороны, при таком подходе обнаруживается множество недостатков. Громкая речь производит много шума и может отвлекать окружающих, находящихся на своих рабочих местах, да и сам голосовой ввод требует тишины, так как естественный шум может искажать смысл сказанного. Необходимо сложное программное обеспечение, а на практике нужно искать компромисс между временем, необходимым для обучения, и словарным запасом. На правильность понимания может влиять болезнь, алкоголь, погода и т.д.

Хотя голосовой ввод и привлекателен для многих приложений, особенно для тех, где руки пользователя не свободны, существуют проблемы с распознаванием продолжительной речи. Чтобы понять, почему это так, рассмотрим предложение: "Очень трудно уничтожить хороший пляж". А теперь попытайтесь произнести его вслух несколько раз подряд и достаточно быстро.[5] Сегодня существуют системы распознавания продолжительной речи, но их словарный запас очень ограничен. Эта задача все еще занимает умы исследователей. Большинство систем голосового ввода накладывают достаточно четкие ограничения на то, что можно сказать и как, или требуют много времени для обучения программы распознавать голос отдельного пользователя.

Как и для устройств ввода, существует несколько вариантов устройств вывода, включая экраны, принтеры и динамики. Сети — это устройства ввода и вывода. Устройства только ввода поднимают серьезные вопросы ЧМВ, по большей части не связанные с физической эргономикой.

Виртуальная реальность, которая погружает пользователя в искусственный мир и предоставляет обратную связь через различные органы чувств и средства передачи информации, бросает величайший вызов всем разработчикам пользовательских интерфейсов. К счастью, все взаимодействия в виртуальном мире можно представить как взаимодействия с объектами, удовлетворяющими формальным описаниям. Однако еще не совсем ясно, какие новые психологические факторы связаны с таким погружением. Возможно, многие из основных принципов пользовательского интерфейса придется подкорректировать, и делать это надо будет с особой внимательностью, так как с ними связана немалая опасность как с медицинской, так и с социальной точки зрения.

Исследования показали, что не существует "идеального" устройства ввода или вывода. Какое из них будет самым подходящим и удобным, зависит не только от приложения, но и от конкретной задачи или набора задач, которые надо выполнить. В результате одного неопубликованного исследования, проведенного компанией Logica, было выяснено, что для биржевых торговцев самым лучшим устройством будет клавиатура, а торговец облигациями достигнет большего, пользуясь графическими планшетами для ввода информации. Однако и задачи, и технологии постоянно претерпевают изменения, поэтому непонятно, можно ли применять эти результаты к современному состоянию вещей.

---

[5] Если во время этого упражнения кто-то находился в пределах слышимости, он, вероятно, согласится, что сказанное действительно было очень трудно понять.

Стили взаимодействия

Существует множество вариантов взаимодействия с компьютерной системой. В их число входят меню, формы, командные языки, естественные языки и графические пользовательские интерфейсы. Как и для устройств ввода и вывода, наилучший стиль взаимодействия зависит главным образом от поставленной задачи. Нет общепринятого наилучшего стиля.

Преимущество меню заключается в том, что его можно быстро изучить. Они обычно требуют меньшего числа нажатия клавиш, чем другие стили, и позволяют пользоваться диалоговыми средствами управления. Меню помогают задать структуру диалога и упрощают разработчику задачу устранения ошибок. Однако у них есть несколько недостатков. Дерево меню может быть очень глубоким, так что запомнить его и перемещаться по нему будет достаточно трудно. Это может значительно замедлить работу пользователей. Меню занимают место на экране, они негибкие и для работы с ними нужны высокие характеристики дисплея. Самое важное условие для разработки интерфейса меню — завершение этапа анализа задач.

Интерфейсы, основанные на формах, удобны, главным образом, для ввода данных и направлены на его упрощение. Пользователи таких интерфейсов нуждаются в незначительном обучении. Однако такие интерфейсы не очень хороши для случайных пользователей. Формы также занимают место на экране и не являются гибкими. Положительная сторона состоит в том, что для сокращения времени разработки можно успешно применять генераторы форм (языки четвертого поколения). Это существенно упрощает работу, по сравнению с использованием языков третьего поколения, где весь код нужно писать вручную.

Модальные диалоговые окна, в которых взаимодействие строго ограничено вопросами и вариантами ответа в текущем окне, представляют собой некую комбинацию подходов, основанных на формах и меню.

Командные языки обладают определенной гибкостью и обычно напоминают макросы или языки программирования. Однако общеизвестно, что ими трудно овладеть, их трудно запомнить и легко допустить ошибку.

Самые древние командные языки — это естественные языки. Часто отмечается, что компьютер должен владеть языком пользователя и общаться с ним на английском или каком-то другом естественном языке. Это означает, что пользователям не пришлось бы изучать новые команды, но здесь есть несколько очень серьезных упущений. Естественные языки выразительны, но многословны и неоднозначны. В работе, даже без использования компьютеров, часто пользуются более формальными или структурированными языками. Возьмем, например, словосочетание "вас понял" на военном языке или на техническом жаргоне большинства профессий. Общение на естественном языке неизменно основывается на врожденном общем понимании и на том факте, что при нарушении стандартных правил необходимы пояснения — а это замедляет общение. Эта проблема наряду с понятием контекста в языке означает необходимость применения методик искусственного интеллекта. К несчастью, эти методики еще не совсем развиты. Бытует мнение, что говорящим компьютером может стать только мыслящий компьютер, а это либо невозможно, либо настолько выходит за пределы разумного человеческого машиностроения, что кажется абсурдным [324]. Существуют естественно-языковые интерфейсы к базам данных. Разработаны даже соответствующие коммерческие продукты, но это не решает общей проблемы понимания языка. Предпочтительнее было бы использовать естественно-языковые запросы в ограниченном контексте, основанные на общей модели словаря данных и его структуры.

9.10.2. Основы психологии познания

Чтобы разработать хороший пользовательский интерфейс, необходимо кое-что знать о том, как работает человеческий мозг и тело. В частности, полезно знать, как осуществляется хранение и поиск информации в памяти, как глаз реагирует на цвета, что вызывает усталость и т.д. Физические аспекты человеко-машинного взаимодействия часто называют эргономикой, хотя в строгом смысле сюда входят также и аспекты, связанные с мышлением.

Согласно современной теории психологии познания, **длительная память** (long-term memory) отвечает за хранение знаний и информации. Чтобы воспользоваться этой памятью, ее нужно активизировать, а это занимает некоторое время. Обычно она находится в неактивном состоянии, хотя не обязательно расположена отдельно от кратковременной памяти, которая доступна немедленно. **Рабочая (активная) память** (activated memory) ограничена и работает недолго. Доступ к элементам рабочей памяти прямой и быстрый. Активность связей этой памяти со временем ослабевает. В [560] отмечено, что активная память может удерживать от пяти до девяти элементов или *фрагментов*, а потом в некотором смысле переполняется. Этот предел зависит от вида информации, кроме того, фрагмент памяти специалиста может содержать намного больше информации, чем такой же фрагмент новичка. Было бы грубой ошибкой считать, что, например, на экране не может быть одновременно представлено больше 7±2 элементов. Например, когда перед новичком находится изображение шахматной доски, он видит 32 клетки. Мастер высокого класса, однако, может видеть только пять вариантов игры высокого уровня, которые он усвоил за долгое время практики. Более того, информация является структурированной (она классифицирована и систематизирована), что упрощает ее поиск. **Многократное обращение** способствует помещению информации в длительную память и доступу к ней. Часто используемые пути активизации запоминаются в нервной системе человеческого мозга. Еще один способ более быстрой активизации этих путей — это **связывание**, когда вызов одного элемента помогает активизировать другой, семантически с ним связанный. Связывание помогает активизировать понятия в рабочей памяти (РП). Когда пользователь переходит от интерфейса одной программы к интерфейсу другой, повторение и связывание вызывают **эффект переноса**. Этот эффект может быть как позитивным (полезным), так и негативным (вредным), с точки зрения возможности использования. Чем больше разработчики пользовательских интерфейсов знают об особенностях памяти, тем лучше они могут выполнять свою работу.

В число других особенностей памяти, которые могут оказаться важными, входят помехи, которые встречаются, когда связывание приводит к активизации не того, что нужно. Это пример негативного эффекта переноса. Позитивный эффект переноса состоит в способности пользователя классифицировать свои знания. Этому помогает постоянство и связность интерфейса. В целом, позитивный эффект проявляется, когда разработчик копирует структуру существующих и хорошо известных задач. Успех представления рабочего стола можно объяснить очевидностью такого подхода. Этим же объясняется популярность объемных клавиш элементов управления, которые вдавливаются, если щелкнуть на них. Классический пример негативного эффекта — это использование клавиши <F3> для вызова справки в программе WordPerfect, тогда как в большинстве пакетов используется <F1>. Когда человек, работающий в WordPerfect, нажимает <F1>, выполняются нежелательные для него действия, а когда он пытается узнать, какая клавиша отвечает за вызов справки, не находит такой информации. Примером применения позитивного эффекта служит поддержка синтаксиса Lotus в программе Excel.

Структурированность упрощает понимание и связывание понятий. Это легко можно проследить по рис. 9.21. На рис. 9.21, *а* близкие по смыслу элементы расположены рядом, а линии подчеркивают это. На рис. 9.21, *б* такая организация отсутствует и меню выглядит запутанным и трудно запоминается.

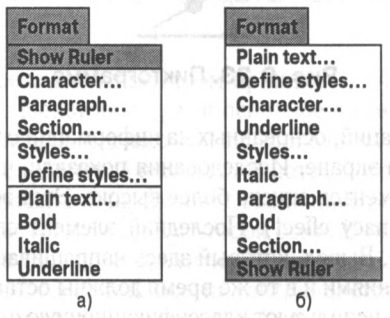


Рис. 9.21. Структурированное (а) и неструктурированное меню (б)

Эффект связывания встречается между близкими словами. Например, слово "кошка" связано со словом "собака", а шрифт *italic* может быть связан со шрифтом **bold**. Приведем еще один пример удачного применения эффекта связывания в пользовательских интерфейсах. Он проявляется, когда для закрепления знаний о свойствах элемента используется изображение (часто это визуальная подсказка или пиктограмма). Рис. 9.22 иллюстрирует этот пример. Там показаны два различных меню для выбора начертания шрифта в текстовом процессоре. Меню справа, очевидно, помогает пользователю запомнить, как выглядят шрифты `Courier` или `Times Roman`. Однако это может зайти слишком далеко. Пользователь, незнакомый с греческим алфавитом, может быть смущен тем, как выглядит шрифт `Hellenic` в этом меню. Эффект связывания зависит от контекста, и разработчики должны понимать это. Например, в контексте медицинской диагностики слово "кошка" должно быть связано со словом "рентген", а не со словом "собака".

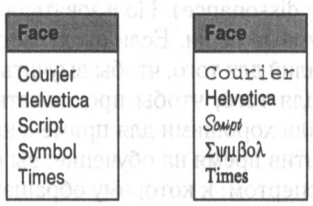


Рис. 9.22. Напоминание о начертании шрифтов

Этот пример поднимает вопрос, неизбежно возникающий при использовании символов, пиктограмм и мнемосхем. Восприятие пиктограмм зависит от культуры пользователя и от основных знаний, которыми он должен обладать. Рассмотрим в качестве примера символ, изображенный на рис. 9.23. Вы или сразу же поймете, что он значит, или никогда не отгадаете

этого. А дело вот в чем. Большинство читателей из Европы или Америки старше 35 лет пользуются устройством с таким изображением. Ответ вы найдете дальше в этой главе. Обычно очень хорошо подкреплять пиктограммы словами.

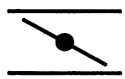


Рис. 9.23. Пиктограмма

Есть несколько рекомендаций, основанных на информации о том, как действует пользователь, когда читает данные на экране. Исследования показали, что вероятность сохранения в ДП первых прочитанных элементов списка более высока. Этот эффект известен под названием **эффект первенства** (primacy effect). Последний элемент списка обычно сохраняется в РП — это **эффект свежести**. Вывод, который здесь напрашивается, — интерфейсы должны быть согласованы с приложениями и в то же время должны оставаться неизменными. В стандартных интерфейсах обычно используют классификационную структуру памяти.

Стандартность пользовательского интерфейса важна из-за ограниченной памяти пользователей. Для меня главное преимущество GUI — это то, что большинство приложений работает одинаково, и я могу воспользоваться позитивным переносом, когда закрываю окно или вызываю меню Help, потому что эти элементы расположены одинаково. По моему мнению, это важнее, чем просто простота использования.

Еще один важный принцип разработки пользовательского интерфейса, порожденный пониманием психологии, — это использование **близости**. Нам всем знакома ситуация "я пришел сюда за чем-то, но не помню за чем". Это происходит из-за человеческого свойства удовлетворения выполнением задания, как только мы приближаемся к его выполнению в каком-либо отношении. Нам свойственно избегать выполнения дополнительных задач. Вот почему большинство банкоматов предлагает нам забрать карточку до того, как мы заберем деньги. При иной последовательности существует вероятность ошибки: как только вы получаете деньги (завершение основной задачи), то можете уйти, забыв карточку и квитанцию. К несчастью дворников, такой принцип редко применяется еще и по отношению к квитанциям. Одна из моделей CREWS, которые обсуждались в главе 7, основана на понятии близости.

Когда для объяснения плохой разработки выдвигаются надуманные причины, происходит **диссонанс в познании** (cognitive dissonance). Пользователи придумывают объяснение тому, почему нужно выполнять неразумные вещи. Если вы делаете что-то достаточно часто и вам пришлось приложить немало усилий для того, чтобы выучить, как это делается, вам придется частенько придумывать повод для того, чтобы продолжать поступать именно так. Таким образом, плохие разработки стали хорошими для применения, а недоделки превратились в отличительные признаки. Потратив время на обучение, вы становитесь источником знаний о продукте. Вы можете быть экспертом, к которому обращаются другие, даже если появится намного качественнее разработанный продукт, для которого не нужно никакого специального мастерства. Мне часто приходилось наблюдать такую ситуацию с малопонятными операционными системами универсальных вычислительных машин, ненужными языками программирования, такими как язык RPG, и макроязыками, такими как Lotus 123 и подобными ему. К широко известным примерам этого явления, вероятно, можно было бы частично отнести стандартную клавиатуру.

Рассмотрев информацию о психологических принципах на очень высоком уровне, можно приступить к определению некоторых практических принципов и рекомендаций по разработке ЧМВ.

9.10.3. Принципы разработки средств человеко-машинного взаимодействия

Во взаимодействии человека с компьютером принимают участие компьютеры, пользователи, задачи и требования. Некоторое время основное внимание уделялось разработке, ориентированной на пользователя. Однако сейчас широко распространилось мнение, что пользователь не является инвариантной величиной, потому что при взаимодействии с компьютерной программой пользователи выполняют различные роли. Постоянным признаком часто выступает задача, соответствующая некоторой роли. Внимание в этой главе и во всей книге фокусируется на проектировании, ориентированном на задачи. В ЧМВ используются принципы кибернетики, психологии, социологии, антропологии, эстетики и многих других наук. В этой книге акцент делается на понимании психологии, разработке программного обеспечения и обработке информации. Предпринимаются также попытки учесть влияние всех остальных областей знаний. Главный вывод инженерии знаний состоит в том, что пользовательский интерфейс содержит в себе знания пользователя о программе и программную модель пользователя. Личное понимание того, что большинство пользовательских интерфейсов ужасно, сильно влияет на это представление.

Критическое значение имеет то, что разработчики пользовательских интерфейсов, как и разработчики программного обеспечения в целом, не должны останавливаться на анализе и автоматизации существующей ручной работы. Задачи компьютера могут изменяться, что происходит достаточно часто. Электронная обработка текста сильно изменила саму природу работы во многих офисах, а развитие "всемирной паутины" продолжает эту тенденцию.

Разработка ЧМВ предполагает решение следующих вопросов.

- Функциональность. Как интерфейс помогает пользователям выполнять поставленные задачи и как он мешает этому? Делает ли сам интерфейс что-либо возможным или невозможным?
- Эстетика
- Приемлемость
- Структура
- Надежность
- Эффективность
- Удобство поддержки
- Расширяемость
- Стоимость
- Удобство применения (легкость в изучении, запоминаемость, возможность совершения ошибок, поддержка задач (анализ задач), безопасность, диапазон пользователей, применимость в различных условиях и ситуациях)

Объектно-ориентированное программирование обеспечивает удобство поддержки, расширяемость и некоторые аспекты удобства применения. Составление прототипов рассматривается многими как существенный элемент производства пригодных для использования интерфейсов. Некоторые авторитетные источники советуют включать в состав команды разработчиков специалистов по графике или дизайнеров, а некоторые идут еще дальше, предлагая совершенно отдельную разработку пользовательского интерфейса (UI) в рамках жизненного цикла. Я думаю, что это было бы ошибкой и что разработчики программного обеспечения должны владеть разумными методами разработки программного обеспечения и объектно-ориентированными методами, которыми в качестве специальных навыков владеют разработчики интерфейса. Здесь, правда, существует опасность, что мои замечания будут восприняты так, будто разработчики могут создавать интерфейс только на основе прецедентов, что в действительности предлагает метод RUP. Конечно, прецеденты действительно **определяют** типы взаимодействия, но, кроме этого, нужно **спроектировать** каждое взаимодействие. Чтобы сделать это, разработчик должен стать специалистом по разработке интерфейса пользователя. Так как GUI играет важную роль, владение такими навыками должно стать нормой для разработчиков; если вы не можете сделать это самостоятельно, вам придется нанимать специалиста.

Замечательно, что наилучшими свойствами GUI часто называют эстетические. Одно из самых популярных новшеств в оконных системах — это объемная кнопка, которая вдавливается, когда на ней щелкают мышью или когда ее продолжают удерживать в нажатом состоянии. Это не добавляет никакой функциональности, но очень нравится пользователям, настолько, что это стало сегодня обязательным для всех программ.

Общие принципы ЧМВ формулируют достаточно редко, хотя несколько рекомендаций все-таки предлагается. Корпорация IBM в 1991 году предложила набор удивительно четких и полных рекомендаций по обеспечению общего доступа пользователей CUA, которые совместимы с объектно-ориентированным подходом. Этот подход подразумевает анализ задач, обзоры пользователей, посещение Web-узлов и тестирование применимости, а также включает построение трех моделей, представляющих взгляды пользователя, разработчика и программиста на интерфейс и программу, лежащую в его основе. В подходе CUA особое внимание уделяется уменьшению нагрузки на память пользователя, согласованности и участию пользователя в разработке интерфейса. В частности, для поддержки последнего принципа авторы советуют не ругать пользователей за ошибки, а учитывать их и предотвращать. Это обеспечивается немедленной обратной связью, приспособлением к пользователям разных уровней, наличием полезных подсказок, соответствием интерфейса требованиям заказчика и его прозрачностью. Нагрузка на память уменьшается в результате применения выразительных интерфейсов классов для управления взаимодействием и приведения конкретных примеров везде, где это возможно. В подходе CUA для обеспечения согласованности применяются принципы эстетики, непрерывности и связывания при помощи четких стандартов и визуальных представлений. Для решения этих задач недостаточно традиционных объектно-ориентированных процессов, таких как FDD, RUP и Perspective.

GUEP

В [757] вводится понятие обобщающих принципов GUEP (Generative User-Engineering Principles) как для разработчиков, так и для пользователей. К этим принципам относятся следующие.

- Распознавание и применение принципа близости.
- WYSIWYG (What You See Is What You Get) — вы получаете то, что видите.
- Утверждение документации, а затем, согласно ей, создание программы.
- Разработчик должен быть в состоянии объяснить интерфейс кратко, но исчерпывающе.
- Разработчик должен быть уверен, что пользователь сможет составить соответствующую внешнюю, мысленную модель системы.

В работе [757] представлен ряд формальных или алгебраических принципов GUEP, в том числе следующие.

- *Идемпотентность* [$T=T^2$]. Некоторые действия при повторном выполнении не должны производить никакого эффекта. Например, если второй раз щелкнуть на кнопке возврата в главное меню, это не даст никакого результата. При удалении файлов повторное нажатие клавиши удаления не должно приводить к удалению следующего файла без особых распоряжений на этот счет. Команды "вырезать", "копировать " и "вставить" также должны быть идемпотентны. Идемпотентность особенно важна при использовании буферизованного ввода с клавиатуры, так как при отсутствии синхронизации могут выполняться нежелательные действия.
- *Дистрибутивность* [$A*(B+C)=A*B+A*C$]. Согласно GUEP, дистрибутивность операций относительно их аргументов может повысить удобство использования, уменьшая объем вводимой информации. Например, команду "удалить файл 1; удалить файл 2" можно заменить командой "удалить файл 1, файл 2". В проводнике MS Windows команда *удаления файла* распространяется на результат команды *выделения файлов*, так что можно выделить пакет файлов, а потом удалить их одним щелчком мыши.
- *Коммутативность* [$A*B = B*A$]. Этот принцип гласит, что порядок выполнения операций не должен играть роли. Например, перемещение курсора (кроме движения на границах экрана) с помощью клавиш <←>, <→> может не привести к изменению его положения. Этот принцип противоречит общепринятому современному нововведению, согласно которому допускаются круговые движения мыши, при которых мышь, выйдя за пределы экрана, появляется на его противоположной стороне.
- *Подстановка.* Это означает, что постоянную можно заменить выражением, как это можно сделать в большей части языков программирования или крупноформатных таблиц.
- *Ассоциативность* [$A*(B*C) = (A*B)*C$]. Этот принцип равнозначен отсутствию модальности.

Режимы в пользовательском интерфейсе определяются как "переменная информация в компьютерной системе, которая отражается на значении того, что пользователь видит и делает". В **режимных** системах пользователь должен знать, в каком режиме работает программа, потому что в разных режимах одно и то же действие может иметь совершенно различный эффект.

Существует общепринятое мнение, что режимов следует избегать, но иногда они необходимы. Например, все согласны, что наличие режимов *пользователя* и *новичка* становится в своем роде правилом. Конечно, программа должна информировать пользователя о текущем

режиме, и очень хороший способ привлечения внимания к переменам — изменение цвета индикатора режима, строки текущего состояния или какого-то другого подобного элемента. Приведем несколько примеров режимов интерфейса: нажатие кнопки ON на обычном калькуляторе, для которого существует два режима — *on* (включен) и *off* (выключен). Результат такого нажатия в каждом случае будет различным. Сообщение "Нельзя сохранить файл — диск не форматирован" свидетельствует об определенном режиме. Это же касается сообщения "Записать поверх существующего файла?". Диалоговые окна обычно модальны в том смысле, что при их появлении недоступны другие опции. Таким образом изменяется поведение части экрана. Понятие режимов или модальности тесно связано с понятием полиморфизма. Полиморфные интерфейсы обязательно модальны. Однако то, что полезно для языка, не обязательно хорошо для интерфейса.

Очень полезно предоставлять пользователям информацию о контексте, в котором будут интерпретироваться их действия. Сюда относится информация о зависимостях между состояниями и режимами. Кроме того, должно быть описание текущей задачи с ее предусловиями, постусловиями и побочными эффектами. Режимами нужно пользоваться только тогда, когда есть абсолютная необходимость ограничить свободу действий пользователя. Временными режимами можно пользоваться для запуска определенной последовательности операций, но при этом нагрузка на память пользователя увеличивается. Этого никогда не следует делать, но если другая последовательность действий может привести к возникновению проблем, то это может иметь место. Самый известный пример — это злоупотребление каскадными модальными диалоговыми окнами там, где лучше работает простая форма для ввода данных. Пространственные режимы обычно применяют, когда возможно выполнение нескольких действий, но не одновременно. Выделение пиктограмм и изменение указателя мыши — хороший способ обратить внимание на изменение режима. В строке текущего состояния можно найти другую контекстную информацию, такую как функция клавиши <Insert>, находящейся в раздражающей близости от главной клавиатуры на большинстве ПК.

Очень мудро поступают те, кто вводит операции *отменить* и *повторить* для поощрения исследовательской деятельности по изучению программы и защиты пользователей от непоправимых ошибок. Однако операция *отменить* может быть несовместима с безмодальностью. Это показано в [757], хотя аргументы автора слишком сложны и запутанны, чтобы приводить их здесь. Главным образом так получается из-за того, что отмена множества действий может каким-то иным способом изменить состояние системы. Современные интерфейсы обычно отвечают "нельзя отменить", когда просчитывают эту ситуацию.

В [757] предлагаются принципы GUEP использования режимов. При этом режимы делятся на общие, инерционные, режимы ввода и вывода. Предлагается также общий принцип **равных возможностей**.

- *Общие режимы*. У пользователя должна быть возможность делать что угодно и где угодно (не должно быть вопросов, предполагающих ответы Да/Нет, или неподвижных диалоговых окон).
- *Инерционные режимы*. Удачный вид экрана должен претерпевать как можно меньше изменений. При возвращении пользователя в программу все должно остаться без изменений. Должны быть выделены используемые [в последний раз|вероятнее всего|часто] команды меню.

- *Режимы ввода.* Система должна быть полностью приспособлена к использованию при выключенном дисплее (например, клавиша <q> или комбинация клавиш <Alt+F+X> для выхода).
- *Режимы вывода.* Что видишь на экране, то и получишь при печати.
- *Равные возможности.* Выход может стать входом и наоборот.

Хорошо известный пример равных возможностей — это запрос по образцу QBE (Query By Example); широко распространенный метод запросов в реляционных или псевдореляционных базах данных. При запросе по образцу пользователю предлагается форма, подобная изображенной на рис. 9.24. Затем пользователь может вписать конкретные значения `Dept` и `Salary`, а QBE выдаст все записи, которые соответствуют образцу. Это принципиально не отличается от запросов на языке Prolog, ответы на которые также получаем в результате сравнения с образцом. **Равные** возможности существуют потому, что пользователь может выбрать **любое** значение; он не ограничен строго определенным, разработанным заранее диалогом.

```
Enter Dept: Sales
Enter Salary: >10,000
<list of matching records>
```

Операции *вырезать* и *вставить* обычно тоже предоставляют равные возможности или, по крайней мере, должны предоставлять.

Name	Dept	Salary
?	Sales	>10,000

Рис. 9.24. Запрос по образцу в системах баз данных

В [757] приводится пример калькулятора, подобного тому, который показан на рис. 9.25, *а*. Простые калькуляторы не обеспечивают равных возможностей, так как результат операции и операнды не взаимозаменяемы. Например, можно ввести `13*4 =` и получить (если повезет и батарейки окажутся хорошими) `52`, но нельзя ввести `?*4 = 52` и получить ответ `13`. Кстати, интерфейсы таких устройств не обеспечивают обратной связи. Если вы захотите проверить, действительно ли 0*0 = 0, то с одинаковым успехом можете решить, что устройство не работает или что законы алгебры подтверждаются.

На рис. 9.25, *б* показана усовершенствованная разработка, основанная на принципах GUEP. Выберите символы для двух кнопочных дисплеев и введите любые два числа, а на третьем появится соответствующий результат. Эту разработку можно усовершенствовать еще больше, если в качестве показателя неисправности дисплея не применять 0.

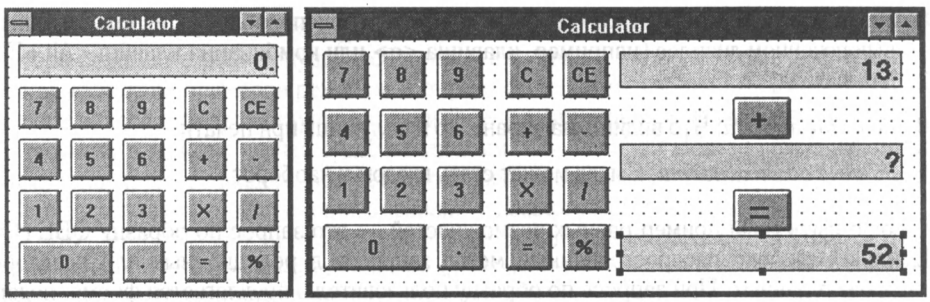


Рис. 9.25. Простой калькулятор (а) и калькулятор с равными возможностями (б)

Остальные принципы разработки, на которые хотелось бы обратить внимание читателя, таковы.

- **WYSIWYG**. Этот принцип уже упоминался без объяснения. Он говорит о том, что форма вывода информации должна соответствовать ее представлению на экране. Например, должны соответствовать расположение, шрифт и размер символов. Символы на экране должны выглядеть точно так же, как и при печати.
- **WYSIWYCU** (What You See Is What You Can Use) означает **что видишь, тем и можешь пользоваться**. Если на экране есть объект, пользователь должен иметь право что-то делать с ним или с его помощью, и это что-то должно быть как можно естественнее и очевиднее. Я надеюсь, что в будущих компьютерных программах этот принцип будет обобщен до WYKAIWYCU (What You Know About Is What You Can Use) — **о чем знаешь, тем и можешь пользоваться**. Я знаю о возможности вырезать и вставлять. Почему я не могу воспользоваться этим в обычном режиме диалогового окна?
- Принцип **соизмеримых усилий** утверждает, что удалить объект должно быть так же сложно, как и создать; отменить какое-либо действие настолько же трудно, как и выполнить его. Этот же принцип выражает лозунг Стива Джоба (Steve Job): "Простые вещи должны быть простыми, сложные вещи должны быть возможными".
- Интерфейс должен обеспечивать **ощущение процесса**. Для этого необходима обратная связь, потому что только так можно связать причины и следствия, и это позволит пользователю видеть, как влияют результаты каждого шага на общую задачу. Таким образом, нужно визуализировать выполнение медленных задач или задержки в сети, чтобы пользователь мог определить, действительно ли программа не работает или просто медленно выполняет задание. Превращение указателя мыши в песочные часы — очень полезный и часто используемый метод, но можно все-таки поинтересоваться, не зависла ли система. В этом отношении очень помогает приблизительная оценка времени ожидания в виде индикатора прогресса.
- **Непредупредительность**. Это хороший принцип, потому что он ограничивает гибкость интерфейса и диапазон задач, к которым его можно применять. С другой стороны, иногда предупредительность действительно необходима для предотвращения катастрофичных, непреднамеренных ошибок. Пользователь случайно может удалить

несколько страниц или непреднамеренно записать информацию поверх других файлов, забыть сохранить какую-то важную информацию, нечаянно отключить клавиатуру, выключить программу или случайно прервать сеанс связи. Для таких опасных действий нужны предупреждающие сообщения.

- **Самодемонстрируемость**. Она подразумевает контекстную справку, обучающие элементы и — самое лучшее — полностью интуитивный интерфейс, основанный на эффекте переноса из удобного представления.
- **Пользователь должен сам устанавливать опции**. Например, драйвер Microsoft Mouse предлагает опции перевода указателя на противоположный край экрана и сохранения его положения на кнопке OK при появлении диалогового окна. Обе эти опции полезны, но для опытного пользователя такое непривычное "ползанье мыши" не будет соответствовать его ожиданиям. К счастью, эти опции можно отключить.
- **Предупреждение частого переключения внимания**. Очевидно, что частое переключение внимания от клавиатуры и мыши нежелательно. Следует избегать отвлекающих сообщений из других частей экрана, а не с той, на которую непосредственно направлен взгляд, а также слишком частой перемены точки концентрации внимания.

Уделив так много внимания принципам, поговорим теперь о некоторых практических рекомендациях по разработке пользовательского интерфейса.

9.10.4. Рекомендации по разработке пользовательского интерфейса

В контексте пользовательского интерфейса и пользователь и программа должны выполнять свои обязанности по отношению друг к другу. Эти обязанности основываются на известных прецедентах. В число обязанностей пользователя входит: знать, какие задачи можно решать, уметь выполнять процедуры, необходимые для решения всех задач, понимать и интерпретировать сообщения (в том числе в различных режимах) и уметь пользоваться соответствующими устройствами ввода-вывода. К обязанностям программы относятся: помогать пользователю выполнять задачи, заложенные при разработке, правильно реагировать на команды, предупреждать о вводе информации, которая может повредить программе, отвечать ограничениям и иногда объяснять себя пользователям. Эти обязанности больше ориентированы на задачи, чем на пользователя, потому что один и тот же пользователь может выполнять различные роли при решении различных задач. Например, один и тот же пользователь может подходить к программе как руководитель, интересующийся выполнением, или как клерк, вводящий информацию.

Разработчики должны помнить, что пользователи значительно отличаются друг от друга. Пиктограммы (изображения) зависят от культуры пользователя, что было проиллюстрировано на рис. 9.23. Для тех, кто все еще не разгадал значение этого рисунка, скажу, что это изображение воздушного клапана в автомобиле. Догадаться об этом можно, либо всецело полагаясь на память либо зная, что внутри карбюратора есть "бабочка" — устройство, состоящее из плоской металлической пластинки, которая надета на иглу. Она регулирует доступ воздуха к месту горения. Как только вы узнали об этом, значение рисунка стало очевидным. Если вы не владеете такими знаниями, вам придется просто запомнить значение. Более того, пользователи сильно отличаются друг от друга своим зрением и поэтому соответственно будут

реагировать. Кроме этих природных различий в способностях, они могут очень сильно отличаться друг от друга из-за физических недостатков, таких как дальтонизм, отсутствие пальцев, утомляемость, неграмотность, расстройства памяти, глухота и т.д.

Принцип степенной зависимости от практики говорит, что умение зависит от практики по логарифмическому закону (или практика ведет к совершенству). Чем больше у пользователей будет возможностей работать с интерфейсом, тем лучше они научаться это делать. Здесь подразумевается, что этому будут способствовать регулярность и стиль работы. Программы, которыми пользуются нечасто, требуют большего внимания к пользовательскому интерфейсу.

Пользователи приступают к работе с программой с различным уровнем подготовки. Согласно исследованиям психологов, при изучении нового объекта знания вначале сохраняются как **декларативные**, часто в форме правил и объектов, к которым эти правила можно применять, и вспоминаются именно в таком виде. Практика помогает человеку построить ассоциации между отдельными элементами и сформировать связи, основанные на этих ассоциациях; это **ассоциативное знание**. Более интенсивная практика способствует объединению правил в **процедурные знания**, которые обычно уже недоступны для сознания. К ним относятся умение читать или ездить на велосипеде. Поэтому разработчики должны ориентироваться на уровень знаний пользователей и по возможности предоставлять режимы для новичка и специалиста.

Вот несколько широко используемых принципов моделирования пользовательского интерфейса, которые можно добавить к изложенным выше.

- Пользуйтесь строгими, естественными модельными представлениями и аналогиями.
- Придерживайтесь принципа простоты.
- Моделируйте непосредственно объекты предметной области.
- Пользуйтесь семантическими структурами (классификация, композиция, используемость, связность).
- Минимизируйте количество семантических примитивов.
- Придерживайтесь правил.
- Помните, что документация, обучение и знания пользователей — это все части пользовательского интерфейса.

Обратите внимание на то, что эти рекомендации похожи на советы по созданию объектно-ориентированных моделей, которые можно найти в предыдущих главах.

Проектирование диалогов

Проблема всех пользовательских интерфейсов, как графических, так и любых других, — это проектирование диалогов. При этом можно воспользоваться множеством наработок, выходящих за пределы кибернетики. К важным областям относятся анализ дискурса и семиотика. Теория построения речи [46, 692] особенно важна для программного обеспечения коллективного пользования. Она повлияла на подход к моделированию требований в методе SOMA, что обсуждалось в главе 8. В [738] для разработки фотокопировальных устройств применяются антропологические и этнометодологические принципы. В [422] описывается ряд исследовательских попыток формализовать взаимодействие, пользуясь грамматикой командного

языка и языками действий. На практике ни один подход пока не дал никакого значительного улучшения. Вероятно, самое важное, что можно отсюда вынести, — это то, что интерактивное взаимодействие зависит от общего понимания и знаний в одной области. Это уже давно известно разработчикам программ понимания естественного языка. Было бы большим (и удивительным) достижением заставить компьютер понять предложение "время летит как стрела", потому что для этого требуется умение сравнивать. Еще менее вероятно, что та же машина разумно ответит на замечание, что "фрукты летают как бананы". Мы уже использовали идею **грамматики задач и действий** при описании прецедентов и задач в форме предложений стандартной структуры.

Подлежащее — Сказуемое — Прямые дополнения — [*Предлог — Непрямые дополнения*]

Обычно сказуемое означает переход. Подлежащее соответствует либо агенту, либо другому активному компоненту программы.

Диалоги применяются для нескольких целей: с их помощью можно отдавать команды, совершенствовать общую цель и планировать решение задач по ее достижению, передавать информацию, данные или знания, проводить время. Большинство программных диалогов сосредоточено на командах и передаче информации или данных. Команды подразумевают передачу части полученной информации для выполнения упомянутых задач. Следовательно, все взаимодействия, инициированные пользователем, должны давать видимый результат, или подтверждая, что задача выполнена, и предоставляя другую результативную информацию, или докладывая об исключениях. Это происходит за пределами цели прецедента. Другими словами, хорошо разработанная программа *помогает* пользователю. Она также должна уметь *прощать* его ошибки.

Экспертные системы часто обеспечивают объяснения для своих рекомендаций, так что их написание усложняется. Поэтому иногда диалог представляет собой объяснение, которое корректирует пользовательское представление о программе. Если у пользователя сложилась хорошая модель программы, он будет считать ее простой в применении. Хорошая обратная связь, так же как и объяснения, помогает пользователю развивать и совершенствовать свою модель. Если у системы есть модель пользователя, она сможет правильнее отвечать ему. Примеры такой ситуации часто можно встретить при компьютерном обучении, когда программа задает новичкам более простые вопросы, основанные на результатах их тестов, или в программах, в которых есть режимы *специалиста* и *новичка*, включающиеся в соответствующих случаях.

Вот несколько эвристических советов, которые будут полезны при проектировании диалогов.

- Минимизируйте количество действий по вводу.
- Сделайте максимальными ширину/количество каналов ввода.
- Тщательно подбирайте слова и язык.
- Интерфейс должен выглядеть привлекательно.
- Будьте последовательны (в размещении клавиш и т.д.).
- Придерживайтесь принципа безмодальности системы или обеспечьте контекстную обратную связь высокого уровня.

- Желательно обеспечить естественное время отклика.
- Пользователю необходимо видеть, как проходит процесс.
- Простота использования должна иметь первостепенное значение.
- Сделайте так, чтобы ваша программа соответствовала требованиям заказчика.
- Сделайте ее непредупредительной.
- Соблюдайте стандарты.

Хорошая разработка диалога обеспечивает согласованность, информативную обратную связь и простоту исправления ошибок. Активные пользователи должны иметь возможность пользоваться клавиатурными эквивалентами команд ("горячими" клавишами). В системе должен использоваться принцип близости. Возможность *отмены/повторения последнего действия* часто бывает полезной, но может оказаться проблематичной, когда действия нельзя отменять или повторять с одинаковым результатом. Если это возможно, диалогами должен управлять пользователь, и они не должны быть режимно-зависимыми. Разработчик должен стараться уменьшить нагрузку на рабочую память пользователя — для этого можно воспользоваться связыванием, эффектами переноса, близостью и другими средствами. Некоторые разработчики настаивают на том, что важную роль должен играть сам язык или используемые слова. Сообщают, например, что некоторых пользователей-женщин беспокоят слова вроде *abort*. Это проиллюстрировано в табл. 9.3.

Таблица 9.3. Использование языка для сообщений об ошибках

Сообщение об ошибке	Оценка
ABORT: error 451	плохо
Я ужасно извиняюсь, но я обнаружил, что файл, которым вы хотите воспользоваться, уже открыт (ошибка 451)	лучше, но многословно
Ошибка номер 451: файл уже открыт. Эта ошибка обычно происходит, если вы забыли закрыть файл при завершении предыдущей подпрограммы	лучше, но скучновато
Ошибка номер 451: файл уже открыт. Объяснить? <Да/Нет>	неплохо
Оставим это в качестве упражнения для читателя	превосходно

Сообщения об ошибках должны быть однотипными, дружественными, конструктивными, информативными, точными и основанными на терминологии пользователя. Часто бывают полезны многоуровневые сообщения. *Гипертекстовые справочные системы* — это простой способ удовлетворить большую часть этих требований.

Командные языки должны соответствовать единым правилам сокращений, должны быть согласованными и стандартизованными. В качестве контрпримера можно привести множество вариантов команды смены каталога в DOS (CHDIR, CD, DIR и т.д.). Кроме того, в различных операционных системах одного производителя уже стало привычным использование нескольких слов для обозначения одного и того же понятия; например, "каталог-директория" или "help-assist-aid".

Привлечение внимания и использование цветов

Существует множество уловок, при помощи которых разработчики привлекают внимание пользователя к определенному месту на экране, информации или сообщению. В [289] сообщается, что советы, приведенные в табл. 9.4, оказались очень эффективными. Цвет очень полезен для привлечения внимания, но нужно помнить о том, что боковое зрение может быть нечувствительным к цвету. Цветовое решение может помочь сосредоточиться на логической организации экрана, упрощении выявления тонких различий и улучшении эстетики. Пользуйтесь этим разумно. Помните о монохромных экранах: люди, работающие на черно-белом мониторе с приложением, разработанным для цветного, могут не обнаружить некоторых функций. Сейчас эта проблема больше связана с принтерами, а не с мониторами. Остерегайтесь неконтрастных цветовых сочетаний. Дайте возможность пользователям изменять цветовую гамму, и будьте последовательны. Есть еще одна причина избегать необдуманного использования цветов. Очень большое количество людей, особенно мужчин, в той или иной мере подвержены дальтонизму.

Таблица 9.4. Привлечение внимания

Средство	Диапазон применения
Интенсивность	до двух уровней
Размер	до четырех
Шрифты	до трех
Мерцание	2–4 Гц
Цвет	не более 5
Звук	мягкие тона, за исключением предупреждающих сигналов
Символы	маркеры, стрелки, окна, линии

Ниже приведены рекомендации по эффективному применению цвета.

- Пользуйтесь цветом разумно. Используйте не более двенадцати цветов, а для ключевых задач — не более пяти.
- Пользуйтесь цветом для увеличения потока информации. Цвета должны быть семантически связаны с элементами. Изменение цвета почти всегда нужно применять для сообщения об изменении режима, чтобы привлечь к этому внимание пользователя.
- Пользуйтесь цветовыми ассоциациями, такими как красное — опасность, но принимайте во внимание культурные традиции (в Китае белый — цвет смерти). По возможности пользуйтесь стандартными ассоциациями, например цветовым кодированием электрических проводов. Красный и желтый цвета можно использовать для обозначения здоровых тканей в медицинских приложениях.
- Учитывайте ограниченность человеческих возможностей. Глаз обычно более чувствителен к желтому и зеленому, чем к синему или красному цвету. Поэтому последние

хуже подходят для изображения деталей или мелкого текста. Отношение яркости основного изображения и фона должно быть 10:1. Одобряются темные символы на светлом фоне. Чтобы уменьшить утомляемость, используйте для фона разбавленные цвета. Избегайте ярких цветов на краях экрана во избежание эффекта мерцания. Лучше всего для этого подходит серый цвет. Пользуйтесь контрастными цветами во избежание проблем с пользователями, которые не различают цвета.

- Учитывайте задачу, которая выполняется. Начертание имеет значение для текстового представления. Для рисунков важен реализм.
- Рассматривайте цвет как часть интерфейса в целом. Предоставьте пользователю возможность изменять цветовую гамму. Будьте последовательны. Для простоты всегда используйте в меню одни и те же цвета.
- Подумайте о том, как пользователь будет выполнять действия, особенно при использовании мыши. После замены процессора я однажды обнаружил, что "перетаскивание", которым я часто пользовался, стало невозможным — просто потому, что информация прокручивалась слишком быстро, и я не могу зафиксировать точку, куда нужно вставить объект. Для преодоления проблем, связанных с перемещением мыши, модификацией аппаратных средств или прокручиванием экрана, операции должны ускоряться пропорционально времени, в течение которого кнопка остается в нажатом состоянии.

Часто хороший эффект дает составление монохромного прототипа с последующим введением цветов. Далее, не следует загромождать экран. Звуковые и цветовые сигналы должны быть согласованы. Время отклика обычно не должно превышать одной секунды. На значительные изменения нужно указывать при помощи сообщений. Начинающим нужно давать больше времени.

Метрики и тесты на удобство использования

Оценка пользовательского интерфейса очень важна. Важную роль играет рецензирование ЧМВ и рекомендации экспертов. В число других ценных методов оценки входят анкеты, наблюдения и отчеты о сценариях тестирования. В число полезных показателей GUI входит время, которое необходимо затратить на изучение отдельной операции или программы в целом; время, необходимое для выполнения отдельной задачи; средняя частота совершения ошибок пользователем; а также сохранение навыков со временем. Эти показатели подразумевают наличие бюджета для экспериментальной работы и сбора информации. Кроме того, нужно отметить, что существующая система должна оцениваться с учетом значений этих метрик во время установки требований.

При создании продуктов широкого потребления стоит проводить семинары по удобству использования, на которых можно проследить, как пользователи из группы тестирования выполняют обычные действия. Обычно это слишком дорого для рядовых разработок, но может оказаться необходимым для систем, предназначенных для очень широкого использования. Любые наблюдения предполагают необходимость метрик для оценки удобства использования. Возможности изучения можно определить, сравнивая время, которое тратится на выполнение задачи до и после длительного использования. Тестирование на удобство применения легче проводить при помощи вспомогательного программного обеспечения, но наиболее удобно выполнять такое тестирование для готового продукта, когда можно проводить сравнение

с предыдущими версиями. При тестировании на удобство использования в качестве одной из основных методик применяется также анализ задач.

Тесты на удобство в использовании должны учитывать роли пользователей, уровень мастерства, частоту использования и возможные социальные, культурные и организационные различия. Анализ прецедентов подчеркивает центральное значение ролей пользователей. При этом имеются в виду не сами пользователи или их роли, а только сочетание пользователя и выполняемой им роли в лице *исполнителя*. Это понятие объединяет уровень мастерства и организационную роль в единую концепцию. При изучении удобства использования могут различаться уровни умения работать с компьютером и с приложением. В большинстве подходов к ЧМВ применяются простые модели уровня знаний.

1. Новичок (отсутствие знаний)
2. Ученик (знания неполные, усвоенные в виде правил)
3. Квалифицированный сотрудник (знания полные, компилированные и не доступные для сознания)
4. Специалист (знания подвергаются критике и усовершенствованию)

Для проведения тестирования на удобство использования необходимо тщательное проектирование эксперимента и статистический анализ. Поэтому такое тестирование обходится очень дорого. На самом простом уровне необходимо определить категорию наиболее типичных пользователей; например, обычные, хорошо знакомые с компьютером, ученики в данной предметной области, англоязычные и с высшим образованием. Тесты должны учитывать эффекты переноса между различными приложениями. Дополнительная трудность при тестировании GUI возникает из-за его графической природы. В то время как интерфейсы командной строки можно протестировать при помощи средств тестирования, которые сравнивают текстовые выходные данные, при тестировании GUI часто приходится сравнивать битовые образы выходных данных. Это усложняется необходимостью пренебрегать незначительными различиями, такими как конечное положение указателя мыши, что придает особое значение тщательному анализу тестов. Подытожив сказанное, можно сделать вывод, что учет удобства применения при проектировании может привести к большой экономии.

Анализ задач

Одна из самых важных методик, доступных разработчику пользовательского интерфейса, — это анализ задач. Мы активно пользовались им в предыдущих главах. Первые версии анализа задач разрабатывались независимо в США и Британии в 1950-х годах. И в той, и в другой стране главное внимание уделялось мотивации и необходимости преодолеть устаревший послевоенный стиль работы и недостаток мастерства. Британцы сосредоточились на иерархическом разложении задач и систематизации человеческих умений, тогда как американцы сфокусировали свое внимание на таких вопросах, как психометрическое тестирование. Реальные задачи раскладывались с учетом психологических возможностей познания, движения и решения задач. Модель человеческого познания на тот момент была достаточно примитивна и не позволяла анализировать сложные задачи, такие как программирование.

Задача — это вид деятельности или набор действий, осуществляемых агентом, выполняющим определенную роль. Этот вид деятельности приводит систему в целевое состояние. Цели могут иметь подчиненные цели; они достигаются выполнением процедур (наборов действий). Для постановки целей на определенном уровне требуется некоторое умение. Они не

должны быть ни слишком абстрактными, ни слишком специфическими. В решении задачи участвуют объекты, действия и классификационные структуры. Иногда цели очевидны. В других случаях цель не известна изначально или может изменяться в ходе выполнения задачи при появлении новых факторов. Разработчики UI должны учитывать в своих методах человеческий фактор, роли, организации и технологии. Цель — это желаемое состояние и задачи, которые необходимо решить для его достижения. Например, моей целью может быть утоление чувства голода. Соответствующая задача — поесть. Сюда можно включить подчиненные цели (или предполагаемые задачи), такие как нахождение пищи, пережевывание и т.д.

Существует два вида задач. **Внутренние задачи** — это те задачи, которые пользователь должен выполнить для работы с программой (приобретение знаний, нажатие определенных последовательностей клавиш и т.д.). **Внешние задачи** — это те, которые пользователь выполняет при работе с программой (интерпретация отчетов, составление документов и т.д.). Выполнение внутренних задач делает возможным выполнение внешних.

Так как имеется два вида задач, существует и два подхода к проектированию. Во-первых, можно попытаться просто поддерживать каждую внешнюю задачу, т.е. обращать внимание только на прецеденты. Этот подход не ставит под сомнение основные бизнес-процессы. В [54] такой подход сравнивается с созданием "интеллектуальных записных книжек". Во-вторых, можно попытаться построить модель, расположенную ниже уровня задач и обеспечивающую некоторое понимание того, что делается и почему. Если существующую задачу можно промоделировать, то эту модель можно рассмотреть критически и оптимизировать. Как мы уже видели, это может привести к предложениям по усовершенствованию бизнес-процессов. В тех случаях, когда задача — это не просто последовательность простых внешних задач, построение моделей внутренних задач равносильно моделированию знаний. Здесь могут пригодиться навыки проектировщика баз знаний. Если задача обычно подразумевает коллективное взаимодействие, как в случае проектирования с помощью CASE-средств или групповой работы, тогда информация о некоторых внутренних задачах может стать внешней. Обычно такие системы моделировать труднее всего, и Барфилд решительно подчеркивает (как я считаю, очень правильно), что это подтверждается большинством CASE-средств, которые обычно решают слишком много тривиальных задач, при этом иногда пропуская важные.

Если задачи являются центральными, типичными и представительными, на них должны обратить внимание разработчики программы. **Центральность** означает, что задача критична для достижения цели. Если задачу упоминают большинство пользователей, она вполне может оказаться центральной. **Представительность** (representativeness) означает близость модели к предметной области. **Типичность** — это когда знания задачи характерны для большинства пользователей.

Анализ задач состоит из трех основных видов деятельности (которые грубо можно считать этапами). Это определение целей программы, выявление задач и, наконец, проведение анализа задач — поиск знаний и умений, необходимых для выполнения задачи. Опять же, здесь существует аналогия с экспертными системами и инженерией знаний. Обычно у инженера по знаниям есть набор более или менее формальных методик выявления знаний, в который входят такие методы, как сам анализ задач, тематический анализ, решетка Келли, методы сортировки, шкала оценок, подсчет частот, профили умений и представления о производительности. Некоторые из этих методик описывались в главе 6, когда мы имели дело с методами определения объектов. В число остальных методов входит извлечение информации из данной области знаний, интроспектирование, проверка результатов, наблюдение (прямое и непрямое), структурированные и целенаправленные опросы, анализ протоколов (параллельный и ретроспективный), "мозговой штурм", анкетирование и анализ результатов обучения

протоколы требуют, чтобы специалист комментировал процесс выполнения задачи пользователя (этот метод еще называют обратным обучением). Прямое наблюдение подразумевает физическое присутствие наблюдателя, тогда как при непрямом наблюдении используются видеозаписи. И то, и другое отнимает много времени и может быть очень обременительным. Параллельные, тогда как ретроспективные протоколы — это отчеты, предоставленные после выполнения задачи.

При проектировании Web-узлов решающее значение для пользователя имеет время; ему не всегда хочется ждать, пока загрузятся все причудливые рисунки, прежде чем он получит возможность поиска или передвижения по узлу. Процедура еще более удлиняется, если пользователь работает через модемное соединение.

Пользовательские интерфейсы и объектные системы

В главе 7 описывался шаблон Model View Controller (MVC), который является ключом к разработке UI. Этот шаблон позволяет интерпретировать почти любой программный интерфейс. Способ сохранения текста в текстовом процессоре отличается от способа его воспроизведения. Модель может дать пользователю доступ к некоторым переменным, но блокирует остальные, если их изменение, скажем, может привести к нарушению целостности модели. Большая часть GUI использует обработчики событий для фиксирования действий мыши и клавиатуры. Это можно смоделировать при помощи контроллера, который посылает сообщения методам приложения. При разработке интерфейсов использование этой модели приводит к разделению понятий на относящиеся к внешнему виду объектов и их основному состоянию. Это не улучшает интерфейс, но придает ему большую структурную гибкость.

Вопросы исследований

В [422] исследуется грамматика командного языка CLG (Command Language Grammar) и другие, еще не доказанные, формальные подходы к анализу задач. В [571] предлагается вертикальная модель взаимодействия в переменных вход-выход. CLG описывает интерфейс как состоящий из трех элементов двух уровней абстракции. Каждому концептуальному компоненту соответствует своя задача и семантический уровень. Так составлена иерархия задач, а объекты и процедуры, обеспечивающие выполнение каждой задачи, описываются на языке в стиле Lisp. Коммуникационному компоненту соответствуют синтаксический уровень (командный язык) и уровень взаимодействия (физические действия). Наконец, физический компонент связан с пространственным и методологическим уровнями. Этот метод по большей части не обоснован.

В языке действий, предложенном в [656], для формального описания пользовательских интерфейсов применяются грамматические правила, обеспечивающие основу для измерения и сравнения. Это многообещающий подход, но он также еще не обоснован. Грамматика действий, предложенная в [422], дополняет описанный выше метод путем включения в него понимания психологии познания и идеи о моделировании пользователя. Эта работа развивает основанную на правилах модель возможностей пользователя.

Программные модели пользователя [816] рассматривают его (пользователя) как один из процессоров или объектов системы. Они обращают внимание разработчика на вопросы удобства использования и познавательные аспекты проектирования и предназначены для измерения удобства использования. Этот подход имеет много общего с объектно-ориентированной разработкой пользовательского интерфейса.

Один из самых старых формальных подходов к проектированию UI — это метод ЦОМП (цели-операторы-методы-правила выбора) (GOMS — Goals, Operators, Methods, Selection rules) и теория сложности познания [140]. Этот подход базируется на задачах и правилах и позволяет прогнозировать время обучения. Он основан на довольно устаревшей модели человеческого познания, которая предполагает существование трех подсистем: перцепционной, моторной и познавательной. Модель построена на уровне этих примитивных знаний. Вначале мы определяем цель или последовательность задач (метод), ведущую к цели высшего уровня. Каждая задача выполняется при помощи перцепционных, моторных или познавательных операций, таких как увидеть пиктограмму, сделать движение пальцем, вспомнить пароль. Правила выбора определяют, когда нужно пользоваться определенными методами. Например, книгу в библиотеке можно найти посредством последовательного поиска или по номеру. Проблема таких подходов заключается в том, что они сложны, занимают много времени и требуют участия квалифицированного психолога. Описания задач ЦОМП могут быть чрезвычайно многословными. Более того, ЦОМП подразумевает, что существует прототип, который можно измерять, так что этим методом нельзя пользоваться на ранних стадиях установки требований. Фактически большинство подходов, которые обсуждались в этом разделе, полезны для оценки, а не для изначального определения пользовательского интерфейса. Метод ЦОМП внес существенный вклад в развитие теории анализа задач. Однако его недостатком является незрелое понимание принципов познания. В [422] предлагается комбинация идей из анализа задач и экспертных систем. Этот метод, получивший название анализа знаний о задачах, обращает внимание не только на задачи, но и на знания пользователей о них. Это многообещающая область для исследований, которая заслуживает дальнейшего изучения. Метод, которым пользуемся мы, намного проще. Он основан на представлении задач в виде простых предложений, описывающих действия над объектами.

Стандарты GUI

Различные рекомендации от компаний Apple, IBM, Microsoft, NeXT, OFS и Sun основаны на объектно-ориентированном подходе к проектированию пользовательских интерфейсов. Следовательно, при условии построения корректной и выразительной объектной модели эти рекомендации можно применять для выбора стандартных компонентов представления объектов в модели. Это особенно легко сделать при наличии библиотеки объектов интерфейса. Большинство стандартов языка Java, например, инкапсулировано в библиотеках Swing. В [558] содержатся подробные указания по разработке пользовательских интерфейсов в контексте MS Windows. Компании Apple, Sun и Open Systems Foundation опубликовали свои рекомендации по стилю интерфейса пользователя, определяющие стандарты для приложений Mac [37], Open Look [740] и Motif [609]. Причем в качестве образца рассматривается документация Apple. Большинство из этих стандартов содержит множество подробностей, включая стандарты размещения элементов управления (например, кнопка **Cancel** должна находиться на 3 мм ниже кнопки **OK** или справа от нее), стандарты типов и схем управления (таких, как показаны на рис. 3.9), стандарты для меню и соглашения об именовании.

Несмотря на терминологические расхождения, во всех стилевых рекомендациях существует нечто общее. Все они позволяют отображать одно или несколько окон, которые делятся на подокна или окна-"потомки" и применяются для одновременного отображения различных частей одного объекта, файла или документа. Для получения дополнительной информации или контекстных подсказок могут также применяться модальные или немодальные диалоговые окна. Окна содержат объекты, над которыми можно выполнять действия, и панель меню,

уточняющую, какие действия допустимы. Стилевые рекомендации согласованы в таких вопросах, как использование одинаковых слов для обозначения пунктов меню. Большинство выдвигает требование наличия строки текущего состояния в нижней части экрана и "затенения" недоступных пунктов меню. Все они поддерживают кнопки, переключатели, флажки, текстовые поля, полосы прокрутки и некоторые виды иерархических меню. Только немногие рекомендации включают счетчики, списки, комбинированные окна, иерархические прокручивающиеся списки, прокручивающиеся меню и т.д. Но даже при использовании этих элементов обычно предлагается обеспечивать иной способ выполнения соответствующих операций. Иногда полезнее увидеть настоящую первоклассную разработку, чем прочитать рекомендации по стилистике.

В этой области существуют также и международные стандарты. ISO 9126 охватывает процессы оценки UI, а ISO 9241 относится к требованиям безопасности и эргономичности дисплеев. Последний нужно дополнить требованиями к удобству применения. Существует также директива Европейского сообщества по здравоохранению и безопасности, которая относится к использованию программного обеспечения.

Необходимо сделать одно предостережение, касающееся стандартов интерфейса. Эти документы содержат стандарты по таким вопросам, как размещение элементов управления, их цвет и т.д., но по вопросам разработки они в большинстве случает дают лишь рекомендации. Главная ошибка — это нежелание отходить в своем проектировании от стандарта. Любое отклонение можно рассматривать как возможное усовершенствование, если оно, по крайней мере в общих чертах, соответствует руководству.

9.11. Тестирование

Мы уже много говорили о тестировании при описании процесса в разделе 9.7. Там было указано, что наиболее полезные рекомендации содержатся в работе [417]. Было показано также, что тестирование должно выполняться на основе прецедентов. Осталось сделать всего лишь несколько дополнительных замечаний.

Тестирование объектно-ориентированных систем, как и обычных программ, состоит из верификации и ратификации. Верификация означает удостоверение в том, что код отвечает спецификации, а ратификация подтверждает, что результаты отвечают текущим требованиям. Тестирование обычно происходит в порядке, обратном порядку создания программы. Таким образом, сначала проводятся элементарные тесты модулей или классов, в конце — тестирование системы в целом, а в середине — интегральные тесты законченных уровней. Регрессионные тесты позволяют оценить, поддерживаются ли функции исходного продукта в последующих версиях. Обычно для автоматизации регрессионного тестирования создаются системы тестов. Серверы нужно проверять раньше, чем клиентские объекты, потому что недостатки серверов могут передаваться их клиентам.

В [417] указывается на то, что полиморфизм усложняет подсчет количества путей через программный модуль, что, в свою очередь, усложняет процесс проверки завершенности тестирования. Любая последовательность щелчков мышью — это потенциальный путь. Решение — использовать для представления возможных путей применения системы прецеденты. По возможности пути развития событий нужно объединять в классы, которые можно тестировать как один путь. Еще одна проблема заключается в том, что унаследованный метод может не работать в подклассе, даже если он работает в суперклассе. Это означает, что нужно

провести вертикальный интегральный тест всей структуры. Такая ситуация может быть связана с тем, что в подклассе используются значения атрибутов, отличные от тех, которые использовались в суперклассе. Возможно, возник конфликт множественного наследования. В [622] продолжено обсуждение модульного тестирования объектно-ориентированных систем.

При использовании объектно-ориентированного языка программирования тестирование должно выполняться во время кодирования, так как над созданием каждого класса обычно работает отдельная команда. Это относится и к интеграционному тестированию каждого уровня или целой системы с добавлением новых классов. Таким образом, тестирование по большей части завершается к моменту перехода на стадию тестирования, которая предназначена только для выявления неожиданных эффектов в системе, работающей как одно целое.

Очень желательно создание системы тестирования, которая должна включать набор приложений и диалогов с известным поведением. Сделайте это, как только пользователям будут предоставлены прототипы. Для проектирования системы тестов можно воспользоваться обратной связью с пользователями. Это действительно окупится очень быстро.

Придерживайтесь всех традиционных методов тестирования (см. например, [581]). Остаются полезными диаграммы состояний и таблицы решений. Главное отличие объектно-ориентированного тестирования состоит в том, что модульные тесты начинаются на более ранних этапах жизненного цикла.

Так как в рамках объектно-ориентированного подхода больше внимания уделяется интерфейсу объекта, чем его реализации, можно ожидать, что успешной окажется методика тестирования по принципу "черного ящика", когда для построения теста используется только спецификация. Это не подтвердилось экспериментально. При исследовании, проведенном в компании Hewlett-Packard [270], оказалось, что метод "прозрачного или белого ящика" позволяет обнаружить большее количество недостатков. Это подразумевает, что разработчики и программисты классов должны активно использовать метод "прозрачного ящика", так как независимые специалисты по тестированию могут быть незнакомы с тем, как работают низкоуровневые функции. В данном случае модель жизненного цикла подразумевает такую последовательность действий.

1. Написание кода
2. Тестирование методом "белого ящика"
3. Создание библиотеки классов
4. Тестирование методом "черного ящика"
5. Документирование
6. Обучение
7. Тестирование пользователями

9.12. Резюме

В этой главе дается обзор современного состояния дел в области организации объектно-ориентированного процесса разработки. Здесь также описаны конкретные методы управления объектно-ориентированными проектами. Основные моменты, которые нужно запомнить, сводятся к следующему.

- Процесс — это существенная часть истинно объектно-ориентированного метода.
- Обычные "каскадные" процессы не подходят для итеративного, инкрементного стиля разработки, реализующего объектную методику.
- Для описания **жизненного цикла продукта** необходимы спиральные модели, такие как DSDM и RUP. Но было бы ошибкой не заметить, что модели жизненного цикла продукта и процесса **различны**.
- Помимо краткого описания жизненных циклов продукта, автор подробно представил полную модель **жизненного цикла процесса** и привел ее пример. В процессах такого типа должна учитываться синхронизация параллельных видов деятельности. Это достигается построением объектной модели видов деятельности и задач проекта. Для обсуждения синхронизации использовалось модельное представление передачи сообщений с учетом пред- и постусловий. При описании управления проектом автор придерживается терминологии стандарта ISO.
- **Программа** состоит из проектов. **Проект** состоит из объектов видов деятельности, операторами которых выступают **задачи**. Задачи одного вида деятельности могут приводить в действие (или останавливать) задачи других, если удовлетворяются их постусловия. Таким способом можно описать модель жизненного цикла любого процесса. С каждой задачей связан свой уровень обязательности (О/Р/Ф/З/Н). Для задач также определены связи с соответствующими методиками. Например, задача подтверждения, относящаяся к виду деятельности по запуску проекта, в качестве рекомендуемой может включать методику изучения отношения затраты/преимущества.
- Мы рассмотрели различные методики настройки процесса разработки: повторное связывание видов деятельности; изменение рекомендаций; переопределение уровней и т.д. Также обсуждалось использование шаблонов для упрощения настройки процесса.
- Метод Catalysis рекомендовался в качестве шаблона (или микро-процесса) для спецификации и разработки компонентов и системы в целом. Было показано, как этот процесс соотносится с жизненным циклом продуктов в рамках DSDM и RUP и как можно использовать идеи методики XP.
- Процесс включает всесторонние подходы к управлению повторным использованием, тестированию, формированию метрик и разработке пользовательского интерфейса. Последнее основано на принципах психологии познания и теории разработки.

Мы также охватили вопросы оценки объектно-ориентированного программного обеспечения и проектов. После подробного рассмотрения принципов и практических советов по разработке пользовательского интерфейса были описаны ключевые моменты тестирования объектно-ориентированного программного обеспечения.

9.13. Дополнительная литература

Идея шаблонов процесса разработки независимо друг от друга предложена в [29, 30, 204]. Метод RUP лучше всего представлен в [463], но в [416] приводится более детальное описание. В [670] добавлено очень много информации об управлении процессом RUP и о таких

вопросах, как оценка. Эти три книги не во всем соответствуют друг другу. Метод DSDM очень хорошо описан в [726], а XP — в [63]. В число остальных книг по XP входят [65] и [421]. Истоки модели жизненного цикла процесса, описанного в этой главе, лежат в методах SOMA [327] и OPEN [334]. В [382] показано, как процесс OPEN (с учетом процесса, описанного в этой главе) можно реализовать в программных средствах, в частности в Process Continuum.

В [295] описан многократно испытанный авторами метод проведения инспектирования программного обеспечения и документации. Книга содержит несколько полезных шаблонов.

Книга [374] представляет неплохое описание состояния дел в области метрик объектно-ориентированных систем.

Книга [688] долгое время считалась классикой по разработке пользовательского интерфейса. В [472] обсуждается искусство разработки пользовательских интерфейсов вообще и некоторых методик в частности. Автор этой работы написала несколько статей по некоторым аспектам разработки систем с использованием передовых технологий UI, включая виртуальную реальность. Самая интересная работа Лаурел связана с применением принципов греческой драмы к проектированию интерфейса. В [738] для разработки пользовательского интерфейса применяются антропологические и этнометодологические принципы. Очень рекомендую эту книгу. Работа [422] связана с анализом задач и знаний. Работа [757] вносит очень важный вклад в изучение вопросов разработки пользовательского интерфейса. Многие принципы, представленные в этой главе, были взяты из этой работы. Работа [595] — это существенное подспорье для разработчиков пользовательских интерфейсов, а [54] — разумный вклад в литературу о ЧМВ. Юмористический стиль автора только усиливает впечатление от замечаний о хорошем проектировании. Наряду с работами [757, 422] она составляет основу для прекрасного знакомства с предметом. В число других полезных публикаций входят [147, 758, 769]. Последняя работа написана одним из основателей интерфейса Apple Mac. [477] дает логически завершенный и хорошо структурированный подход к разработке GUI, основанный на объектно-ориентированных принципах. Ударение делается на анализ задач и принципы разработки интерфейса. В [477] описывается, как изучать удобство применения. Автор основывает свой метод на анализе задач в рамках традиционной модели "цели-операторы-методы-правила выбора". Он упрощает эту модель посредством выражения задач в виде простых предикатных предложений и хорошо объясняет свой подход на примере административного управления персоналом. Представлены особенно четкие рекомендации по использованию модельных представлений пользовательского интерфейса, таких как рабочий стол, и кратко описано проектирование структуры GUI. В [124] предлагается метод, удовлетворяющий традициям SSADM. Он напрямую не относится к разработке интерфейсных элементов. Однако работа содержит некоторую полезную информацию. Еще один процесс разработки UI, основанный на принципах, изложенных в этой главе, — это GUIDE [652]. В [48] приводится полный список источников по теории разработки пользовательского интерфейса, включая многие из тех, которые предшествовали появлению графических интерфейсов. Если вы начинаете проект по разработке ЧМВ, вам может быть полезно почитать соответствующие учебники и рекомендации. Классическая ссылка — руководство от компании Apple [38]. Вдохновляющими могут оказаться рекомендации из [588]. Разработчикам систем для продуктов компании Microsoft понадобится книга [558].

Работа Майерса [581], посвященная тестированию программного обеспечения, все еще сохраняет свою важность.

9.14. Упражнения

1. Почему компании стараются стандартизировать процесс разработки? С какими препятствиями они могут столкнуться?
2. Перечислите преимущества временны*х* блоков.
3. Приведите примеры метрик для продукта и процесса. Как они связаны?
4. Дайте определение термина "модель оценки". Приведите пример такой модели.
5. Дайте определение низкого зацепления методов (НЗМ). Постройте диаграмму НЗМ для какой-нибудь реальной разработки. Какими математическими свойствами обладает этот показатель? Таковы ли они, как ожидалось?
6. Дайте определение ветвления.
7. Дайте определение следующим терминам:
   а) ключевая задача;
   б) элементарный прецедент;
   в) существенный прецедент;
   г) функция.
8. Дайте определение трех метрик объектной модели задачи и четырех метрик объектной модели бизнес-процесса.
9. Опишите главные задачи двух из следующих видов деятельности:
   а) планирование временны*х* блоков;
   б) построение временно*г*о блока;
   в) оценка;
   г) планирование реализации
10. Сколько времени проходит от проведения семинара по установке требований до реализации?
11. Какая разница между образованием и обучением? Что более важно?
12. Нарисуйте каскадную, спиральную, контрактную модели жизненного цикла и модель RUP.
13. Назовите восемь ролей в проекте. Подробно опишите две из них.
14. Закончите фразу: Пользователи должны быть _____, но не слишком _____.
15. Какая разница между последовательностью выполняемых действий в технологии RUP, видом деятельности и задачей в терминологии стандарта управления проектом ISO?
16. Составьте меню, аналогичное изображенному на рис. 9.22, и обеспечьте подсказку без использования греческих символов.
17. Обсудите использование пиктограмм в графических пользовательских интерфейсах, включая их преимущества и возможность неправильного использования.

18. Должен ли указатель мыши двигаться дальше, если рука пользователя двигается быстрее? Если нет, то почему?
19. Назовите семь принципов, которыми следует руководствоваться при разработке пользовательского интерфейса; в каждом случае укажите, почему это важно, и приведите примеры, когда их невыполнение приводит к плачевным результатам для пользователя.

10

Приложения

Такого никогда не видели у нас, и я боюсь.

В. Шекспир. *Юлий Цезарь*

Эта глава представляет собой обзор некоторых традиционных и новых приложений объектно-ориентированных методов программирования, а также обращает внимание читателя на некоторые существующие и возможные ограничения. Почти все, что выполняется с помощью обычных вычислительных методов, можно сделать и при помощи объектно-ориентированных методов. Многие объектные приложения уже описывались в этой книге, но мы еще раз вернемся к самым характерным из них, включая те, которые до сих пор встречались с определенным сопротивлением. Затем пересмотрим некоторые прогнозы на будущее, сделанные в более ранних изданиях этой книги, и проверим их точность, обладая преимуществом ретроспективного взгляда в прошлое, чтобы понять, где были допущены просчеты или была проявлена непредусмотрительность.

10.1. Web-приложения

Приложением, которое заставило практически все организации в мире принять объектные технологии, стала "всемирная паутина" World Wide Web со своим новым спутником — электронной торговлей. Уже говорилось о том, что технологии, как животные, подчиняются закону естественного отбора. Новая вариация может выжить лишь в случае идеально подходящих для нее условий. Объектные технологии (ОТ) прозябали, пока новая бизнес-модель электронной торговли не заставила бизнес-структуры, охотнее чем отделы ИТ, подумать об использовании языка Java и серверов приложений. Если бы такое суперприложение встретилось на пути развития искусственного интеллекта, его история была бы совершенно другой. Электронная торговля построена на многих взаимосвязанных технологиях: сетевые, межсетевые соединения, базы данных, программное обеспечение среднего уровня и объектно-ориентированные методы. Кроме того, она требует реорганизации структуры компаний.

История Internet начинается в 1969 году с проекта ARPANET: это была реакция министерства обороны на осознанную угрозу ядерного разрушения коммуникаций в условиях холодной войны. Созданная система выжила, потому что ее пользователи — военные и научные круги — нашли чрезвычайно полезными такие ее особенности, как электронная почта и средства FTP. Она основывалась на пакетной коммутации, а не на коммутации каналов, так что сообщения содержали информацию о своем маршруте и могли таким образом направляться по наиболее подходящим путям в случае неполадок в сети или ее узлах. Подобная система, разработанная в то же время в Национальном институте исследований в области компьютерной обработки данных и автоматики (INRIA) во Франции, оказалась менее жизнеспособной, возможно из-за меньшего количества пользователей. Тем не менее именно Европа породила World Wide Web, когда Тим Бернерс-Ли (Tim Berners-Lee) и его коллеги из Европейского центра ядерных исследований CERN в 1990 году выдвинули идею гипертекстового броузера. Протокол HTTP поддерживал графику поверх протоколов Internet — TCP/IP, SMTP и т.п. — и привел к созданию World Wide Web.

До World Wide Web и протокола HTML существовало несколько гипертекстовых систем, таких как Hypercard от компании Apple, работающая на автономных компьютерах. Термин "гипертекст" относится к идее, впервые выдвинутой изобретателем мыши Дугом Энглбартом (Doug Englebart) еще в 1963 году [259, 260]. Подобные идеи еще раньше, в 1945 году, предлагал военный советник Ванневар Буш (Vannevar Bush). Само название начало употребляться намного позже, в работах [583, 584], когда машины стали достаточно мощными. Интерактивная система ссылок и телеконференций XANADU, разработанная Нельсоном, и сейчас доступна под UNIX. Гипертекстовые системы преследуют две главные цели: улучшение управления комплексными системами доступа к информации и усовершенствование пользовательского интерфейса. Идея, скрывающаяся за системой XANADU, имела еще большие перспективы. Нельсон представлял ее как огромную, соединенную с международными сетями общедоступную базу данных, которая может отвечать на любые запросы пользователей: текстовые, графические, видео или звуковые. Эта идея была полностью реализована только в World Wide Web. Благотворительный проект Gutenberg Project призван сделать доступной в редактируемой электронной форме всю мировую литературу, что выходит за рамки авторского права. Он еще далек от завершения, в чем я убедился, пытаясь перепроверить цитаты, приведенные в этой книге, но уже является довольно-таки полезным источником.

Гипертекстовую систему можно рассматривать как набор текстовых фрагментов, соединенных сложной сетью указателей. Гипертекстовый документ читается не последовательно,

как роман, а согласно содержанию, как энциклопедия. Человек читает текст, отмечает нужное или интересное слово или понятие, а затем может перейти непосредственно к объясняющему тексту, обычно указывая на интересующую фразу или изображения или щелкая на них кнопкой мыши. Такая система работает с различными данными, в том числе с текстовыми, графическими, видео или музыкальными. Таким образом, более общее название "гипермедийный" предпочтительнее, чем "гипертекстовый".

Гипермедийные фрагменты следует рассматривать как узлы или страницы, а указатели — как каналы связи. Узлам, если нужно, можно присвоить определенный тип, чтобы указать предмет обсуждения или вид страницы, скажем текст или графическое изображение. Несколько узлов могут составлять сложный узел, образуя модульную структуру. Связи бывают двух типов: "от" и "к". Единственная связь "от" ведет от данного узла к последнему посещенному, а набор связей "к" регистрирует узлы, доступные в данный момент для следующего шага в путешествии по миру информации. Связь между узлами обычно устанавливает разработчик, который выделяет элемент и записывает, на что он указывает. Некоторые системы создают связи динамически или автоматически. Примером этого может служить оглавление документа или предметный указатель. Исходя из этого, можно с легкостью заключить, что объектная технология сильно влияет на разработку таких систем, а языки объектно-ориентированного программирования часто выступают основой для их развития. Следует заметить, что представление искусственного интеллекта о семантических сетях также оказывает определенное влияние на эту область.

Простые системы просмотра рассчитаны, в первую очередь, на читателя, а не на писателя. Они обеспечивают простоту интерактивного доступа к большим объемам информации, таким как политические или методические руководства, справочные системы или руководства по эксплуатации, где важны удобство и простота изучения и использования. Многие из сегодняшних Web-узлов представляют собой просто "брошюру", описывающую некую организацию, продукт или человека. Более изощренные связаны с базами данных, дающими доступ к информации общего пользования. **Внутренние корпоративные сети** (intranet) делают данные организации доступными для персонала фирмы, тем самым предоставляя возможность перестраивать бизнес-процесс, не подвергая риску конфиденциальную информацию. На многих узлах используются аплеты Java, элементы управления ActiveX или CGI-сценарии, позволяющие пользователям как отправлять, так и принимать информацию. **Внешние сети корпораций**, или **экстрасети (extranet)**, позволяют компаниям безопасно общаться с клиентами, обеспечивая доступ извне зарегистрированным надлежащим образом узлам. Такие узлы, как `amazon.com`, являются образцом электронных торговых отношений между клиентом и компанией (business-to-customer — B2C). Сложность в прогнозировании числа клиентов узла свидетельствует о том, что балансирование нагрузки и работа с буфером с использованием серверов приложений, обычно основанные на объектно-ориентированных базах данных и программном обеспечении среднего уровня, имеют очень важное значение. Торговые отношения между компаниями (B2B — business-to-business) позволяют обеспечить интеграцию сетей и осуществлять электронные платежи и заказы без дороговизны традиционного электронного обмена данными. Компания даже может продать с аукциона свою будущую продукцию человеку, предложившему наивысшую цену.

Пока еще не существует международных компаний, основу функционирования которых обеспечивает Web, хотя банк Citibank уже объявил о своих планах вести дела с 16% мирового населения к 2012 году. Система электронной торговли, основанная на технологии CORBA, уже отвечает за 15% торговых операций на фондовой бирже NASDAQ.

В гипертекстовых системах, допускающих связывание гиперссылок с некоторыми процедурами, естественно возникает вопрос о возможности обработки информации с целью имитации "разумного" прохождения по информационной сети. Здесь пришел на помощь язык XML, но реализация подобного подхода зависит еще и от использования объектно-ориентированного программного обеспечения среднего уровня. Новички в электронной торговле легко могут стать специалистами в области объектных технологий и пользоваться ими, но существующим компаниям необходимо сохранить свое наследие. Здесь необходимым условием успеха является четко определенная структура приложения. Методы, описанные в главе 7, являются основными для осуществления такого стремления. Еще один урок электронной коммерции: плохо разработанный интерфейс более недопустим. Когда я работал над этой книгой, оказалось, что поставщик спортивной одежды `boo.com` вынужден закрыться. Одной из причин послужило то, что не все потенциальные клиенты могли с легкостью попасть на их узел. Другими словами, компания не предусмотрела возможности общения с клиентами по коммутируемым линиям связи, а также с потребителями, не обладающими новейшими технологиями поиска. Рекомендации, данные в главе 9, и методы, описанные в главах 6 и 8, нацелены на решение этой проблемы.

10.2. Другие коммерческие приложения

На протяжении последнего десятилетия XX века приложения объектно-ориентированного программирования развивались быстрыми темпами. Не считая электронной коммерции, самыми активными сферами были финансы, телекоммуникации, производство и сама компьютерная индустрия. Как и все новые технологии, объектно-ориентированное программирование было встречено с определенной долей скептицизма со стороны авторитетных профессионалов в области информационной технологии. Чтобы преодолеть это сопротивление, необходимо было показать удачные примеры приложений. Первые флагманские проекты показали, что можно строить очень большие объектно-ориентированные системы, наибольшие из которых насчитывают около 900 000 строк кода и 3 500 классов, но это убедило не всех.

10.2.1. Графические пользовательские интерфейсы

Почти до 1990 года самым популярным применением объектно-ориентированного программирования была разработка пользовательских интерфейсов. Причина этого — во-первых, влияние языка Smalltalk, в котором особое внимание уделяется пользовательскому интерфейсу и автоматизации офисных операций с самых первых дней существования системы Dynabook, и, во-вторых, сложность типичных графических пользовательских интерфейсов (GUI — graphical user interface). Подход к решению этой проблемы был верифицирован компанией Xerox не только на примере языка Smalltalk, но и InterLisp — их собственного диалекта языка Lisp. Влияние этой первой работы на дальнейшее развитие пользовательских интерфейсов Apple Lisa и Macintosh хорошо известно. Этот пиктограммный стиль пользовательского интерфейса сейчас распространен почти везде.

Создавать графические пользовательские интерфейсы очень дорого. Если для создания интерфейса Apple Lisa понадобилось свыше 200 человеко-лет, большая часть которых пошла на разработку программного обеспечения (а значительная его часть была связана с интерфейсом), то кажется невероятным, чтобы интерфейс Apple мог перейти в интерфейс

Macintosh, а затем в Macintosh II без повторного использования кода интерфейса Lisa. Так как темпы развития компьютерного оборудования ускоряются, поставщики не могут позволить себе через каждые пять лет получать новые результаты, требующие затрат в 200 человеко-лет. Это касается не только разработчиков вычислительных средств, но и пользователей. Большинство современных графических пользовательских интерфейсов просто нельзя описать, не пользуясь объектно-ориентированными методами. Более того, работа с приложениями в такой среде становится просто невозможной без библиотек повторного пользования и расширяемых графических объектов. Простая программа "Hallo world" для ранних версий Microsoft Windows занимала около 83 строк кода на языке C, не считая повторно используемого кода API Windows, и для работы вне Windows требовала редактирования, компиляции и повторной компоновки. Код API в таких системах огромный, и управляемый событиями стиль взаимодействия с пользователем вдвойне усложняет программирование. Программисту, работающему на объектно-ориентированном языке, не надо так усердно бороться с кодом API, так как новые типы окон, полос прокрутки, переключателей и т.п. могут быть определены как экземпляры существующих классов. Объектно-ориентированный или какой-либо другой стиль, поддерживающий повторное использование и расширяемость, действительно необходим для эффективного программирования в такой среде. Объектно-ориентированное программирование также упрощает эксплуатацию и обслуживание системы. Таким образом, существует чрезвычайная необходимость в объектно-ориентированных решениях проблем создания, поддержки и использования графических пользовательских интерфейсов. С помощью хорошей библиотеки классов объектно-ориентированное программирование может ускорить разработку приложений в такой системе, как Windows, на 75%. Язык Visual Basic с самого начала содержал встроенные объекты пользовательского интерфейса с изменяемыми атрибутами и незаполненными методами, которые соответствуют событиям. Например, объекту командной кнопки отвечает метод обработки щелчка, который может содержать код действия, выполняемого при щелчке левой клавишей мыши на кнопке. Дополнительные объекты этого типа могут быть написаны на языке C или приобретены у третьих сторон. Язык Visual Basic является примером того, как объектно-ориентированный стиль можно использовать в качестве среды разработки приложения. Сейчас существует Web-узел с эффективными средствами разработки GUI для объектно-ориентированных приложений, основанных на языке Java или C++.

Объектно-ориентированное программирование полезно не только для разработчиков GUI и авторов приложений, его могут применять и пользователи, чтобы упростить использование приложений для каких-то особых целей. В первом поколении объектно-ориентированных интерфейсов, таких как интерфейс Macintosh I, ни пользователь, ни программист приложения не имели доступа к пиктограммам, как к объектам. В более поздних системах, таких как NeXt или HyperCard, программисты уже имели ограниченный доступ и могли расширять интерфейс. Сегодняшний курс — дать такой же уровень доступа пользователям.

Ранние характерные области применения объектно-ориентированного программирования для развития пользовательского интерфейса включают комплексную систему управления пользовательским интерфейсом UIMS, разработанную для электронных приложений и написанную на языке C++ [647], и визуальный интерфейс баз данных GLAD, написанный на языке Actor [809]. Компания Apple Computer разработала открытую для пользователей систему визуализации Double Vision, которая помогает ученым создавать гибкие цветные двухмерные диаграммы [566].

10.2.2. Моделирование

Моделирование сложных систем было прерогативой объектно-ориентированного программирования с самых первых дней существования языка Simula. Транспортные системы, ядерные реакторы, интегрированные офисные системы, финансовые системы и производственные процессы — все они успешно реализуются с помощью этого метода.

В [591] обсуждаются основные отличительные особенности моделирования объектно-ориентированных офисных систем. Компьютерная игра Sim City — это образовательная система, моделирующая ситуации городского планирования и позволяющая игрокам анализировать последствия различных решений по бюджетным, ценовым, транспортным и экологическим вопросам.

10.2.3. Географические информационные системы

Географические информационные системы (ГИС) — это особые компьютерные системы, заменяющие или автоматизирующие некоторые аспекты аналоговой, бумажной картографии и хранения записей. Обычно они содержат в себе подсистемы для

а) получения данных с уже существующих карт, внешних датчиков, фотографий, сделанных со спутников, и т.д.;

б) хранения и поиска информации в той форме, которая обеспечивает быстрый анализ и обновление данных;

в) систем управления базами данных для работы с информацией об атрибутах определенных пространственных функций;

г) составления отчетов и вывода карты на дисплей.

Если говорить подробнее, типичный продукт ГИС представляет собой такую структуру карт, когда несколько контурных карт накладываются на основную. Такая система осуществляет быстрый доступ к информации о близости полигонов, таких как области или страны. Часто возникает необходимость сохранять данные о полигонах, узлах и каналах связи, а затем осуществлять быстрое планирование, изменение масштаба и создание графических изображений. ГИС позволяет сохранять демографическую информацию, которая может быть полезной в маркетинге или при расчете оптимальных затрат на путешествия. Большая часть ГИС сохраняет информацию двух видов: картографические данные и данные об атрибутах объектов, которые наносятся на карту. Реляционные системы картографических данных представлены достаточно бедно из-за их неспособности сохранять сложные, структурированные объекты. По этой причине картографические данные обычно хранятся в собственной файловой системе. Данные об атрибутах часто хранятся в реляционной базе данных, поскольку этими данными часто пользуются и другие приложения. Во многих приложениях изменения в картографических данных должны отражаться на атрибутах и наоборот. Эта связь является одной из важнейших задач для разработчиков продуктов ГИС. Можно пользоваться реляционными системами, поддерживающими технологию BLOBS, но они не позволяют интерпретировать данные.

История географических информационных систем сравнивалась с историей гражданской авиации [629]. По поводу последней заявлялось, что она состояла всего лишь из трех значительных событий: первого полета братьев Райт, первого гражданского авиалайнера — DC3 — и Боинга 707. Сравнение строилось на том, что первая удачная информационная

географическая система КГИС (Канадская географическая информационная система) нашла успешное применение, в то время как большинству других вовсе не удалось оторваться от земли, как из-за плохого дизайна, так и из-за неудачной реализации программного обеспечения. Это случилось в те времена, когда вычислительными процессами занимались почти исключительно машины, предназначенные для пакетной обработки данных, использовавшие для ввода данных перфокарты, и это уже само по себе было замечательным достижением. В 1970-х годах развитие интерактивных компьютерных технологий сделало возможной революцию в области географических информационных систем, а эквивалентом авиалайнера DC3 стала система ARC/INFO. Ее авторы выбрали для хранения данных-атрибутов существующую псевдореляционную базу данных (INFO) и добавили собственную топологически структурированную базу данных (ARC) для картографических данных. Аналог Боинга 707 еще должен появиться, и мы без риска можем предположить, что это случится в ближайшем будущем по двум причинам.

1. Огромное количество средств, инвестируемых в настоящий момент в эту технологию, которое, судя по всем исследованиям рынка, будет расти.
2. Появятся новые возможности для создания как баз данных-атрибутов, так и картографических баз данных в единой эффективной системе управления базами данных на основе объектно-ориентированных технологий. В частности, современные объектно-ориентированные базы данных сочетают в себе эффективность сетевого подхода, основанного на указателях, с гибкостью реляционного, как мы видели в главе 5.

Можно ожидать, что появится новое поколение ГИС, основанное на одной объектно-ориентированной базе данных, которая обеспечит эффективный стандарт для обмена информацией между различными ГИС. Таким образом, сегодняшнее поколение продуктов устареет по двум главным причинам: отсутствие стандарта для хранения информации и упор на негибкие топологические схемы хранения.

Современные ГИС в большей степени полагаются на автоматизированное проектирование (CAD — computer-aided design) и методики баз данных и обычно работают с нестандартными, собственными системами управления файлами. Чтобы понять причину этого, нужно уяснить, почему невозможно достичь эффективного доступа к топологическим данным из реляционной базы данных. Картографические данные могут быть представлены в различных видах: аналоговом (бумажном), растровом (карта в виде совокупности битов) или в виде векторного изображения. Векторные изображения особенно удобны для хранения сетевой информации на основе одной из двух информационных моделей: "спагетти" или топологической модели. Первая обеспечивает достаточно быстрый доступ, но плохо приспособлена для анализа. Современный продукт почти всегда адаптирован к топологической модели. Такая модель сохраняет не только расположение указателей и строк, но и данные о смежности и пересечениях. Это достигается путем хранения указателей, связывающих топологические объекты (точки, строки и полигоны), и иногда кодирования местоположения равнозначных пар с упорядочением по мере удаленности. Типичный метод — это упорядочение Пеано (Peano), при котором точки сохраняются в виде чисел, образованных чередованием битов двоичного представления долготы и широты точки. Это значительно ускоряет изменение масштаба и расчет расстояния, так как точки хранятся в порядке удаленности. Проблема с реляционными базами данных состоит в том, что им необходимо хранить информацию в нормированных таблицах, и эти таблицы не обязательно соответствуют каким-либо физическим объектам, таким как строки или полигоны. Такие объекты должны создаваться во время

выполнения программы путем объединения нескольких таблиц. К сожалению, объединение — это самая медленная операция, которая может потребоваться от базы данных. При ее выполнении сетевые базы данных могут быть во много раз эффективнее. С другой стороны, сетевые базы данных очень негибкие, и ими трудно управлять в условиях развития схемы. По этой причине данные, содержащие атрибуты свойств карты, такие как ширина дороги, лучше всего хранить в реляционной базе данных, потому что именно эти данные наиболее уязвимы при изменениях в схеме. Картографические данные не столь уязвимы и могут храниться в специально созданных для этой цели сетевых базах данных. Так устроены почти все продукты, начиная с системы ARC/INFO. Объектно-ориентированные базы данных впервые предлагают возможность хранения всей информации в одной базе. Компания Pacific Bell разработала пилотную систему для поддержки сети, используя раннюю версию системы ObjectStore (см. главу 5). Более поздние географические информационные системы, такие как система Smallworld, оптимальны для объектно-ориентированного способа хранения картографических данных. Авторы системы Smallworld решили создать собственную версию языка Smalltalk и объектно-ориентированную базу данных, потому что в то время, когда они начинали, никаких объектно-ориентированных баз данных не было. Производителям, начинающим свою работу сейчас, предоставляется гораздо лучший выбор. Большая часть компьютерной продукции использует стандарт AutoCAD обмена полями данных при хранении карт. Такие системы варьируются от полной реализации стандарта AutoCAD, например в системе Addmaps, до системы Geo/SQL, где для хранения базы данных-атрибутов и их отображений используется продукция корпорации Oracle. Стандарт AutoCAD применяется только для демонстрации и обеспечивает пользовательский интерфейс.

10.2.4. Параллельные системы и аппаратные средства

Параллельными называют процессы, которые происходят в одно и то же время. Если эти процессы происходят вне компьютера, то необходимо найти способ справляться с многочисленными требованиями, касающимися одновременной обработки данных. Многопользовательские операционные системы и стратегии блокировки в базах данных — вот два решения проблем такого рода. Еще один тип параллелизма возникает, когда два или больше процессов выполняются внутри самого компьютера. Это называется параллельными вычислениями. Сначала рассмотрим параллельные процессы.

Модели параллельных вычислений обычно включают передачу сообщений. Это настоящие сообщения, в отличие от метафорического обмена сообщениями в рамках объектно-ориентированного подхода. Однако аналогия настолько сильная, что трудно устоять перед искушением рассматривать объектно-ориентированное программирование как естественное решение проблем параллелизма. Более того, как отмечено в [759], альтернативные модели разделения памяти при параллельных процессах оказались не настолько пригодными для работы с распределенными реализациями, как модели инкапсуляции. Объектно-ориентированные подходы поддерживают кластеризацию объектов, которые часто обмениваются информацией, и это также помогает поддерживать выполнение параллельных задач.

Параллельные объектно-ориентированные системы должны определять, являются ли объекты синхронными или асинхронными по отношению к своим методам. Другими словами, может ли объект или исполнитель продолжать процесс после того, как передаст или получит сообщение. Здесь обычно применяются термины блокировки отправителя или получателя. Понятие передачи сообщения подразумевает передачу информации единым структурным компонентом, тогда как обычные модели разделения памяти при параллельных процессах

этого не делают, а включают в себя использование сложных семафоров. Более того, объекты, соединяющие в себе данные и методы, выступают естественными единицами распределения в распределенных реализациях.

Существует ряд предложений, касающихся языков параллельного объектно-ориентированного программирования, и похожих идей, тесно связанных с объектами, таких как системы исполнителей. Подобно тому, как в языке Smalltalk все сущности являются объектами, в системе исполнителей все сущности — это исполнители. Исполнитель обладает идентичностью и текущим поведением, которое со временем может измениться. Это поведение определяет, как исполнитель ответит на следующее полученное сообщение. Поведение состоит из атрибутов, называемых *знаниями* (acquaintances), и методов, именуемых его *сценарием* (script). Знания определяют других исполнителей, с которыми данный исполнитель может обмениваться информацией. Исполнитель без знаний — это кандидат для "сборщика мусора". При получении сообщения текущее поведение может измениться. Если у исполнителя нет метода для обработки сообщения, он может *делегировать полномочия* (delegate) *посреднику* (proxy) — исполнителю, выбранному из числа "знакомых". Это похоже на понятие суперкласса, но здесь методы не передаются, т.е. код не копируется; происходит лишь передача сообщений. В отличие от объектно-ориентированного программирования, для систем на основе исполнителей существует однозначная семантика, но такие языки относятся к очень низкому уровню и сложны в употреблении.

Кроме исполнительских языков, таких как Act 3, существуют также параллельные версии чисто объектно-ориентированных языков, такие как ConcurrentSmalltalk [813], и гибриды, такие как Orient84/K, сочетающий в себе стили языков Smalltalk и Prolog [409]. Язык Eiffel содержит неразвитые, но полезные возможности параллелизма, а язык Java поддерживает идею потоков, что обсуждалось в главе 3.

Объектно-ориентированное параллельное программирование остается полем исследования, но его огромное значение в будущем гарантируется двумя неопровержимыми фактами. А именно: для достижения эффективности использования ресурсов распределенные структуры предполагают параллельную обработку данных, при этом никакой другой стиль программирования не предлагает такой ясной и принципиально экономичной модели для этого вида сложных систем.

Параллельные вычисления не имеют ничего общего с количеством процессоров в машине. Многопроцессорная машина может параллельно выполнять несколько программ, однопроцессорная — создавать видимость работы нескольких процессоров (используя квантование по времени). Однако многопроцессорная машина обычно может обрабатывать только одну команду программы приложения в каждый такт времени, а ее другие процессоры выполняют специальные операции, такие как операции ввода-вывода.

Существует несколько видов параллельных компьютерных систем. Основными являются структуры с совместно используемой памятью, где применяется крупномодульное разбиение процессов, которые используют один массив данных, и конструкции с распределенной памятью. В системах с распределенной памятью обязанности по хранению данных распределяются по нескольким процессорам, которые обычно реализуют декомпозицию процессов на более мелкие. Структуры с распределенной памятью делятся на два основных вида: либо все процессоры выполняют одну и ту же программу, как машины связей (connection machine) и векторные процессоры, либо разные процессоры могут выполнять различные программы, как машины баз данных, транспьютерные системы или гиперкубические структуры. Нейронные сети являются примером параллельных машин, в которых на низшем уровне все процессоры идентичны, но на более высоком уровне абстракции возникают новые процессы.

Процессоры в системах с распределенной памятью обмениваются информацией, посылая сообщения. Объектно-ориентированное модельное представление, следовательно, непосредственно отражает проблему описания топологии процесса в таких машинах. Из этого следует, что языки объектно-ориентированного программирования и проектирования, похоже, станут стандартом для таких машин.

Начиная с языка Simula объекты были способом реализации как структур данных, так и процессов, в том числе и параллельных. Необходимость описания этих систем подчеркивают распределенные архитектуры, включающие в себя сети и вызов удаленных процедур. Еще раз стоит отметить, что объектно-ориентированное модельное представление используется непосредственно по назначению и, похоже, станет стандартом для приложений этого вида.

Параллельные вычисления все еще остаются проблемой, требующей исследования. Многие вопросы должны решиться еще до того, как технология созреет и получит широкое коммерческое применение. Это ключевая технология, которая будет развиваться в тесной связи с объектным подходом.

Strand

Одну из возможных реализаций параллельных языков программирования высокого уровня представляют языки Strand и Parlog [280] — логические языки программирования в традициях языка Prolog. Язык Prolog основан на идее декларативного представления всех операторов программы при помощи хорновских выражений и применения алгоритмов автоматического вывода во время выполнения, обычно в форме обратной цепочки. Этот процесс можно представить как сопоставление с эталоном или как доказательство теоремы. Хорновское выражение — это особый вид правила логической импликации для одного терма. Например, правило `if A and B then C` записано в хорновской форме, тогда как `if A and B then C and D` — уже нет. На языке Edinburg Prolog первое выражение можно записать в виде `C :- A,B`. Защищенные хорновские выражения (Guarded Horn Clause) — это расширение логического программирования, используемое в параллельных системах доказательства теорем, таких как Parlog или Strand, которое поддерживает параллельное выполнение всех ветвей дерева доказательства. Защищенное хорновское выражение выглядит следующим образом.

$$H :- G_1, G_2, \ldots, G_m \mid B_1, B_2, \ldots, B_n,$$

где H — голова, $\mid$ — оператор подтверждения, $B_1, B_2, \ldots, B_n$ — тело выражения, а $G_1, G_2, \ldots, G_m$ — защитные условия или предварительные условия, при которых может выполняться процесс. Этот стиль программирования объединяет в себе формальную спецификацию и параллельное выполнение.

Одной из привлекательных особенностей языка Strand является его переносимость с последовательных машин на параллельные. В некоторых областях применения принцип, предложенный языком Strand, оказался успешнее объектно-ориентированного программирования. Это касается передачи данных по телефонным каналам [280]. С другой стороны, нет иного такого естественного способа построить модель обмена информацией между процессорами, как объектно-ориентированный подход. В главе 3 было показано, что существует исследовательская инициатива, направленная на объединение функционального, логического и объектно-ориентированного программирования. Следует ожидать, что первые сколько-нибудь успешные приложения этого направления будут относиться к программному обеспечению параллельных компьютеров.

В объектно-ориентированном анализе, где каждому объекту соответствует собственный поток управления, все системы рассматриваются как параллельные. Недостаточная поддержка параллелизма в языках объектно-ориентированного программирования означает несоответствие между анализом и программной реализацией. Параллелизм и потоковые возможности таких языков, как Eiffel и Java, не обеспечивают решения этого вопроса, и я все еще верю, что дальнейшее развитие как возможно, так и необходимо.

10.2.5. Другие приложения

В настоящее время большинство организаций развивают клиентские компоненты базовых систем на основе объектно-ориентированной технологии и все больше и больше основывают свою стратегию развития на объектах. Дело в том, что объектно-ориентированное программирование можно применять почти где угодно. В предыдущих главах мы уже обсудили несколько приложений.

К областям применения объектно-ориентированного подхода относятся системы визуального программирования, CASE-средства, компьютерные интегрированные производственные системы CIM (Computer Integrated Manufacturing), электронные издательские системы, обучающие компьютерные системы и некоторые системы автоматизированного проектирования, где вопросы, касающиеся объектно-ориентированных баз данных, являются первостепенными.

Эти приложения достаточно типичны. Одно из таких приложений — это анализатор блоков ветвления в программах, написанных на языке C [631]. Эта программа, написанная на языке Smalltalk, определяет точки ветвления, т.е. блоки кода, выполнение которых может зависеть от значения одного или нескольких параметров. Она также сохраняет статистику о тех ветвях, которые использовались при контрольном запуске программы. Следует отметить, что для повышения эффективности система была переписана на языках C++ и Objective-C.

Многие из новейших шаблонов проектирования можно возвести к уровню архитектурных шаблонов, например шаблоны классной доски или конвейеров. Параллельно с этим направлением многие современные организации, занимающиеся разработкой программного обеспечения, строят свою системную стратегию на основе служб и брокеров объектных запросов ORB (object request broker). Цель такого подхода — повышение возможностей повторного использования компонентов и увеличение способности системы к взаимодействию с внешними программами. На рис. 10.1 показана схема такой архитектуры. Заметим, что брокер объектных запросов выступает в качестве общего канала, передающего запросы служб к различным серверам на разных уровнях абстракции. На низших уровнях организация надеется купить почти все: службы имен, базы данных и т.п. На верхнем уровне содержится большинство специфических бизнес-приложений. Средний уровень более проблематичен: хотелось бы купить уже готовые бизнес-объекты, но на рынке они практически отсутствуют, да и в любом случае существует специфика работы фирмы, которую не отражают стандартные объекты. В лучшем случае третья сторона сможет обеспечить использование универсальных горизонтальных компонентов, таких как среда разработки графических пользовательских интерфейсов или средства доступа к базам данных. Вертикальные компоненты будет намного труднее согласовать. Таким образом, бизнес-объекты (в виде спецификации или кода) пересекают границу покупка/создание. Заметим, что модель бизнес-объектов, обозначенная пунктирным эллипсом, охватывает как стандартные средства, так и фрагменты приложений, написанные по индивидуальному заказу. Это означает, что внешние серверы вряд ли смогут предложить по-настоящему полезные вертикальные библиотеки бизнес-объектов

в таких сферах, как финансы, телекоммуникации или производство. Такие библиотеки появятся, но они будут содержать только часть нужных классов.

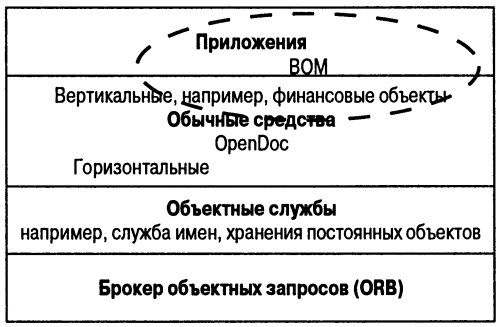


Рис. 10.1. Архитектура программы, написанной с использованием объектной технологии

Альтернативный и более конкретный пример такой архитектуры показан на рис. 10.2. При такой организации стандартные финансовые службы, такие как генераторы кривых, применяемые для ценообразования, и календари, которые определяют, когда открыт рынок, представлены как службы, основанные на технологии CORBA. Обычно это программы на языке C++ с IDL-оболочкой. Пользователи редко имеют прямой или универсальный доступ к этим службам. Специалисты, оценивающие торговые сделки и объемы продаж, могут иметь доступ к этим службам через таблицы Excel или специальные приложения. Другие пользователи могут иметь доступ к тем же службам через Web-броузер.

Брокеры объектных запросов занимают центральное место среди самых первых стратегий перехода к объектно-ориентированным системам. Многие компании также двигаются в направлении агентных технологий. Возникает вопрос: могут ли здесь быть полезными брокеры объектных запросов? Здравый смысл и целенаправленные рассуждения подсказывают, что продукты на основе технологии CORBA должны обеспечить инфраструктуру для агентных систем, включая стандартный язык общения агентов. В настоящее время до этого еще очень далеко, в первую очередь из-за семантических ограничений языка определения интерфейсов IDL (Interface Definition Language), в котором отсутствует концепция наборов правил или хотя бы инвариантов. Однако многие производители ORB, о которых уже говорилось, хотели бы добавить такие функциональные возможности в арсенал своих продуктов, и, если этого будет достаточно, в будущем мы сможем увидеть некоторую стандартизацию их попыток. На данный момент каждая организация, желающая внедрить агентную систему, должна создать собственный язык общения агентов. Совместно с использованием технологии CORBA это имеет большой смысл. Как мы видели в главе 4, при реализации промежуточного программного обеспечения, ориентированного на работу с сообщениями и ORB, обычно используется язык XML.

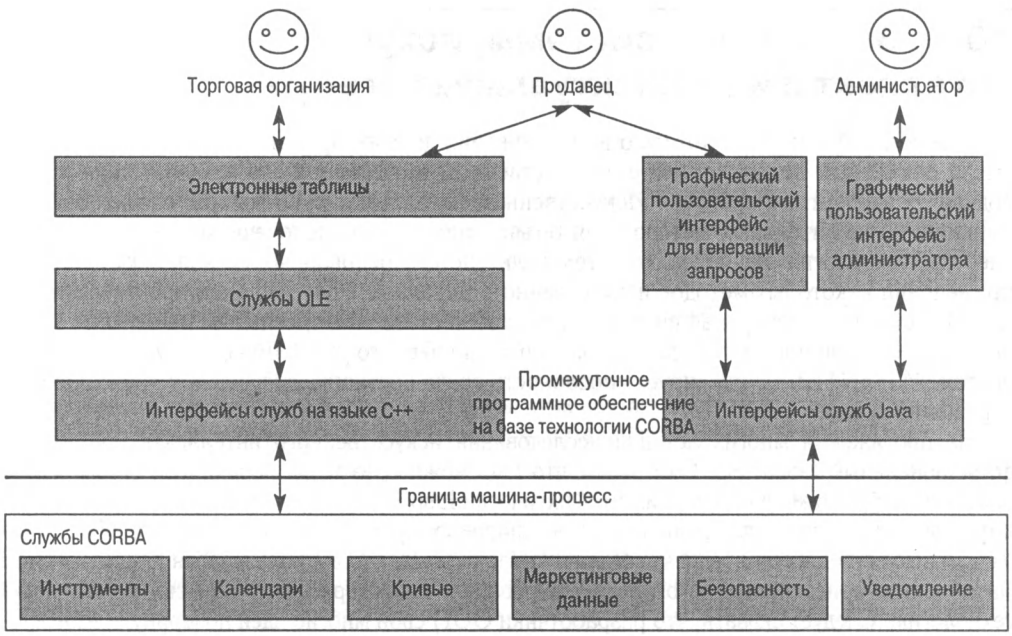


Рис. 10.2. Архитектура торгового приложения

Самые гибкие структуры — это наборы компонентов, подсистемы которых соответствуют бизнес-процессам. Связи между ними соответствуют взаимодействию самих бизнес-процессов. Это дает три преимущества по сравнению с обычными централизованными структурами баз данных.

- Нарушения в работе системы могут повлиять только на одно подразделение, а не на всю компанию в целом.
- Каждую подсистему можно настроить согласно с методами работы местных пользователей. Локальные изменения вносить легче, и во многих случаях можно избежать согласования с центральными отделами информационных технологий.
- При реорганизации бизнес-процессов систему тоже можно перестроить.

Объектно-ориентированные методы можно использовать для организационного моделирования, что обсуждалось в главе 8. Это дает возможность изучать стратегии с помощью моделирования, а затем проверять и сравнивать альтернативные сценарии.

10.3. Экспертные системы, искусственный интеллект и интеллектуальные агенты

Одной из областей, на которую на протяжении долгого времени сильное влияние оказывал объектный подход, является искусственный интеллект и его коммерческая производная — экспертные системы. Искусственный интеллект, грубо говоря, сводится к применению вычислительных методов для объяснения работы человеческого мозга. В свою очередь, технология экспертных систем (или систем, основанных на знаниях) состоит в применении некоторых методов искусственного интеллекта для решения практических задач. В частности, экспертная система не способна стать универсальным решателем задач, но может смоделировать поведение "эксперта" в какой-то узкой области. Одна из первых систем, MYCIN [127], моделирует способность врача поставить диагноз при инфекционном заражении крови.

На протяжении многих лет для исследования искусственного интеллекта применялся функциональный язык Lisp. Оказалось, что Lisp можно без труда расширить, чтобы он мог работать с объектами и сообщениями. Фактически Lisp — это один из самых первых языков, который был расширен для решения самого широкого круга задач. По этой причине исследователи искусственного интеллекта были в числе первых, кто воспользовался преимуществами объектно-ориентированного программирования. Благодаря этому они сильно изменили технологию. Следует сказать, что разработчики ООП свои многие идеи почерпнули из искусственного интеллекта и до сих пор пользуются несколько другой терминологией. В частности, специалисты по искусственному интеллекту используют термин "фрейм". Фреймы [562, 563] были придуманы для представления стереотипов объектов, понятий или ситуаций, но реализация фреймов выглядит подобно объектам в объектно-ориентированном программировании. Особенно это касается свойства наследования. Чтобы понять фреймы, нужно разобраться в архитектуре экспертных систем.

Экспертные системы определяются не только тем, что они делают, но и тем, *как* они это делают. В типичной современной экспертной системе "экспертиза" или "знание" хранится отдельно от стратегии логического вывода, обычно в форме простых правил вида И/ИЛИ. В эти правила можно вносить изменения, и это означает, что экспертные системы легче поддерживать, чем некоторые другие виды компьютерных систем. Это стало особенно важно после того, как обнаружилось, что затраты на поддержку уже существующих систем намного превышают затраты на создание новых. Более того, структура экспертной системы позволяет объяснять пользователю логику действий, что может оказаться немаловажным в том случае, когда человеку нужно действовать наверняка, полагаясь на советы компьютера в критических или очень тонких ситуациях.

Экспертные системы — это попытка представить знания из определенной предметной области в компьютерной системе. Правила хороши для представления причинных знаний, но еще есть и много других видов знаний: знания о процессах, объектах, интуитивное знание и т.д. Например, было бы невыразимо сложно представить знания о форме и свойствах распределителя, используя только утверждения типа "если..., то...". Намного лучший способ — это представление объекта в виде набора атрибутов и методов (объекта и класса объектов). В мире искусственного интеллекта такие объекты иногда называют модулями или сценариями процедурных абстракций, но обычно все же их именуют **фреймами** (frame). Экспертные системы с правилами и фреймами имеют такую же выразительную силу, как и логические языки программирования, но намного проще в обращении, по большей части благодаря

своей способности взаимодействовать с конструкциями высшего порядка и структурами наследования.

Фреймы с самого начала были тесно связаны с понятием наследования (см. главу 3). Фрейм — это структура с расширяемым набором "слотов", которые содержат атрибуты понятия или объекта. Слот может иметь разные **фацеты** (facet), такие как значение, ограничивающие условия, значение по умолчанию и прикрепленные процедуры. Таким образом, методы рассматриваются не столько как интерфейс, сколько как средство для заполнения аспекта значения или для инициирования действия в системе. У некоторых слотов цель особая — определять наследование. Например, фрейм служащего может иметь AKO-слот, содержащий данные о суперклассах, от которых он может наследовать свои свойства. Этот слот также может иметь фацеты, представляющие наследование с различных позиций. Например, с основной точки зрения платежной системы служащий — это вид налогоплательщика, но с позиции зоологической классификации его можно с таким же успехом считать млекопитающим. Таким образом обеспечивается суперпозиция многих невзаимодействующих иерархий наследования для различных целей. В искусственном интеллекте фреймы (объекты) наследуют значения, а в объектно-ориентированном программировании объекты обычно наследуют только способность иметь значение. Фреймы содержат указатели на прикрепленные процедуры, которые тоже наследуются, хотя прикрепленные процедуры не полностью инкапсулированы и могут быть доступны независимо от фрейма. Для искусственного интеллекта множественное наследование является скорее нормой, чем исключением.

Еще одна особенность экспертных систем — это их способность работать с неопределенными или оценочными утверждениями. Это нужно для систем, представляющих знания эксперта-человека. Рассмотрим правило "вкладывать инвестиции в компании, чьи акции сильно упали в цене". Здесь понятие "сильно" не имеет точного значения, но прекрасно понимается. Существует несколько методов управления такой неопределенностью. Один из них, очень популярный в Японии, но менее популярен на Западе, — это нечеткая логика. Но и за другими методами далеко ходить не надо, они включают в себя теорию вероятностей, различные теории доверия и многочисленные специальные и описательные методы. Описанию неопределенности и нечеткой логики в системах, основанных на знаниях, посвящена работа [331]. В приложении А описано нечеткое расширение для объектно-ориентированного анализа.

Подведя итог, скажем, что экспертная система состоит из "базы знаний", содержащей объекты и правила, соответствующие знаниям специалиста о данной предметной области, "механизма логического вывода", представляющего собой программу, которая применяет информацию из базы знаний к данным системы, а также некоторых средств работы с неопределенностью. Если система не содержит специфических знаний и лишь обеспечивает доступ к какой-либо другой базе знаний, то такую систему называют "экспертной системой-оболочкой". Такие оболочки — это высоко продуктивная среда разработки экспертных систем.

Экспертные системы помогают компаниям всего мира в таких разнообразных областях, как страхование, финансовые сделки, получение разрешения на банковские ссуды, контроль над сложным производством, таким как цементные и доменные печи, диагностика дефектов оборудования, оптимизация баз данных, выбор и комбинирование химических покрытий, конфигурация компьютеров и телекоммуникационного оборудования, проверка на честность как в банках, так и в телефонных компаниях, планирование и составление расписаний, определение формул химических соединений, размещение на Желтых Страницах, диагностика неполадок в межбанковских платежных системах и т.д. Еще одна ключевая область применения — это сама обработка данных и оценка проектов (в рамках CASE-средств). Характерная для 1980-х годов крикливая реклама экспертных систем в 1990-х годах была

вытеснена растущим количеством примеров реального применения, большинство из которых стали несомненно традиционными.

Ряд экспертных систем-оболочек, которые представляют собой среду программирования, содержащую встроенные средства логического вывода и методы управления неопределенностью, а также некоторые средства для пополнения знаний, обладает объектно-ориентированными свойствами. Все системы, обсуждавшиеся в главе 3, такие как Aion, ART, Kappa и Nexpert Object, — это среды разработки экспертных систем с большим или меньшим количеством объектно-ориентированных признаков. Их производители сейчас пытаются позиционировать свои системы как "объектно-ориентированные среды разработки" с разной степенью обоснованности и зачастую выпускают их в форме библиотек компонентов для языков Java или C++ (например, система Nexpert). Обычно их сила состоит в реализации наследования, а не в средствах абстракции. Это делает их пригодными для разработки приложений со сложной семантикой процессов или бизнес-правил, в противоположность семантике данных. Таким образом, в результате упорного труда создается ряд приложений, которые обычно, хотя и не всегда, представляют собой экспертные системы. Эти приложения характеризуются не насыщенностью знаниями (хотя и это для них также возможно), а содержанием структур наследования, поведение которых при принятии конфликтного решения отражает важную информацию о самом приложении.

Aardvark

В системе SACIS учитывается соответствие характеристик покупателя свойствам продукта. Политика компании Aardvark — продавать только безопасные игрушки. Это не так просто, как может показаться на первый взгляд. Продукт, безопасный для взрослого (например, снаряжение шпагоглотателя), может быть опасным для десятилетнего ребенка. Игрушку, безопасную для десятилетнего (например, набор шариков), бесспорно рискованно давать в руки младенцу с естественным желанием что-то проглотить. В рамках системы SACIS экспертная система должна содержать компонент, отвечающий за соответствие товара клиенту. Этот компонент работает на основе правил и свойств сложных структур и типов покупателей. Не пытаясь описать все приложение, рассмотрим входящие в него структуры наследования (рис. 10.3).

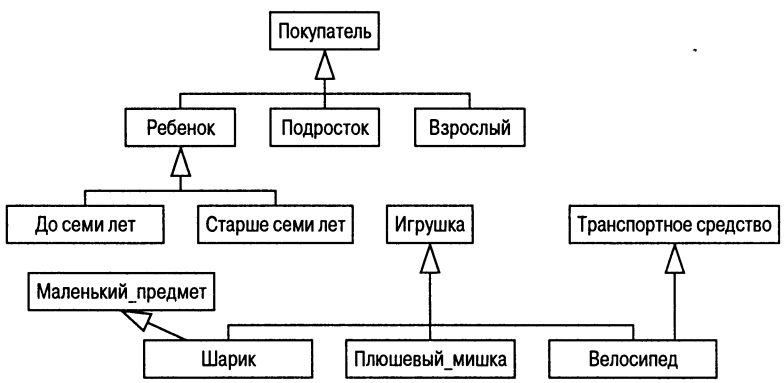


Рис. 10.3. Фрагменты структуры наследования в SACIS

Покупатели делятся на детей, подростков и взрослых, а не просто инкапсулируют возраст. Дети подвергаются дальнейшей классификации — младше или старше семи лет. Значения по умолчанию и ограничения по возрасту или дате рождения инкапсулированы внутри самих структур. Игрушки можно просто разделить по их функциональным свойствам. Заметим, что шарик `Marble` является результатом множественного наследования свойств игрушек `Toy` и маленьких предметов `Small_item`. Маленькими считаются предметы, размер которых меньше 30 мм. Это значит, что шарик наследует свойство быть маленьким и методы, касающиеся манипуляций с маленькими объектами. Теперь правило `if toy is small then safe is false` можно инкапсулировать в объекте `Infant` без воздействия на другие объекты и без применения избыточного кода.

10.3.1. Архитектуры "классной доски" и системы исполнителей

Мы уже встречались с архитектурой "классной доски" в главе 7. В сложных системах, а особенно в системах реального времени, может быть много источников знаний и множество планов и процедур. Типичные области приложения — это системы ведения боя и места совершения финансовых сделок.

Модели "классной доски" применяются в военной среде, где, например, боевой пилот должен обрабатывать большое количество входных данных и делать выбор из ограниченного списка действий. Данная модель — это одна из нескольких видов систем, основанных на знаниях, контролирующая ввод данных и дающая советы пилоту, когда случается что-то интересное, например когда среди фиктивных мишеней или угроз встречается реальная мишень или угроза. Финансовый деятель находится в похожей ситуации, получая большое количество данных из различных источников информации. Все они требуют анализа, чтобы определить, произошло ли что-либо интересное, требующее дальнейшего анализа, и какие действия следует предпринять в этой связи.

Модель "классной доски" получила свое название потому, что эта система напоминает группу высококвалифицированных экспертов, собравшихся у доски для решения определенной задачи. Вновь полученная информация вывешивается на доске, где ее могут видеть все эксперты. Когда кто-то из экспертов видит, что может добавить новый факт, основанный на специализированном знании, поднимает руку. Этот факт может подтверждать или опровергать гипотезу, уже находящуюся на доске, или выдвигать новую. Теперь новые данные доступны для других экспертов, которые, в свою очередь, могут вступить в дискуссию. Руководитель группы управляет экспертами и упорядочивает их предложения согласно плану, вывешенному на доске. Общее хранилище — это доска, а выполнение плана контролируется специальной программой логического вывода. В финансовой сфере эксперты могут быть представлены экспертной системой технического анализа, системой фундаментального анализа, советником по возможным стратегиям и т.д. Если, например, по телеграфу приходит новая информация о ценах, специалист по прогнозированию биржевой конъюнктуры должен определить возможные изменения, но ему нужно дождаться подтверждения, прежде чем дать сигнал. Однако фундаментальному аналитику нужна только малая часть подтверждающей информации, чтобы предположить снижение уверенности в успехе. Набор данных может быть достаточным для выработки нужного сигнала и в результате привести всех специалистов по прогнозированию к успеху. Возможно, действия по продаже также привлекут внимание специалистов по возможным стратегиям, перед которыми встает необходимость пересмотреть свои позиции, чтобы сохранить надежность страхования или избежать других неожиданных случаев в сегодняшних неприятных условиях рынка.

Объектно-ориентированные системы и системы исполнителей дают возможность объектам отправлять и получать сообщения. Модель "классной доски" требует, чтобы доска "выдавала в эфир" свое содержание. Сообщения "вывешиваются" в области общедоступных данных. Объектно-ориентированный подход не допускает такого модельного представления, но здесь есть простое средство. Доску можно разделить на клетки. Каждый агент (или другой объект) регистрирует в специальных клетках свою заинтересованность в ее данных. Когда содержание клетки меняется, агенту передается сообщение с новыми данными или просто флаг "изменения". Во втором случае агент должен решить, следует ли передавать ответное сообщение объекту "классной доски". Агент может подавать на доску информацию о любых существенных изменениях своего состояния: свои "мнения". Архитектура "классной доски" показана на рис. 10.4. Сегодня это понятие широко используется в качестве примера архитектурного шаблона. Здесь мы просто укажем на большое значение этой архитектуры для систем, в которых интеллектуальным агентам приходится объединяться для достижения общих целей или совместного решения какой-либо задачи.

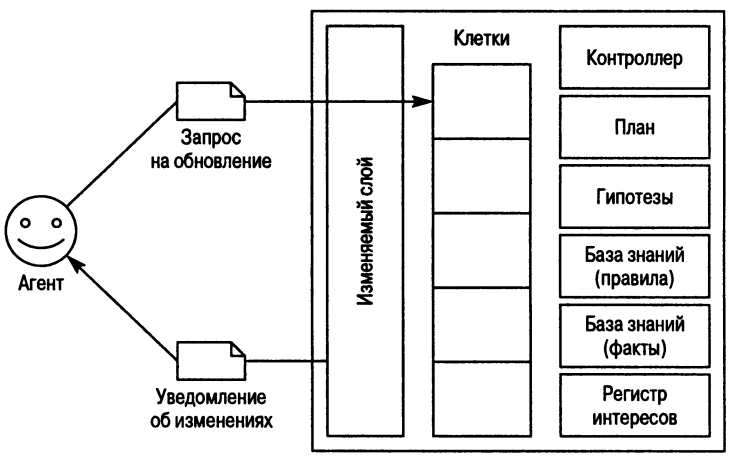


Рис. 10.4. Архитектура "классной доски"

Обычно сообщения рассматриваются в порядке их поступления и в зависимости от состояния объекта-получателя на данный момент. Таким образом, подобные системы обычно имеют вид систем реального времени и выигрывают при реализации на параллельных аппаратных средствах. С учетом этого, одной из самых первых реализаций модели "классной доски" была система Hearsay-II, которая решала проблему понимания речи — единственную задачу реального времени в полном смысле этого слова. С тех пор такие системы находили применение главным образом в военных коммуникациях, системах командования и управления, так что открытых публикаций было мало. Однако системы "классной доски" применялись для управления процессами производства и в системах поддержки принятия решений по сложным денежным операциям. При этом они почти всегда использовали объектную технологию.

BLOBS

В связи с накоплением информации в области общедоступных данных и глобальной доступностью этих данных, системы "классной доски" наталкиваются на проблемы с непрерывными рассуждениями. С другой стороны, системы исполнителей должны заранее знать, с какими объектами нужно будет связываться. Именно по этим причинам были предприняты попытки объединить формализмы некоторых систем. В этом направлении компанией Cambridge Consultants Ltd. развивалась среда разработки программ BLOBS, предназначенная для решения таких задач, как управление положением кораблей. Система BLOBS поддерживает объектно-ориентированный стиль программирования, так что вместо передачи данных, определяющих порядок действий, в программе сообщения передаются объектам. Объекты синхронизированы с таймером реального времени, и "планирующий" объект может быть настроен на обработку расписания. Объект (что не удивительно для системы под названием BLOBS) называется "блобом". Вот пример блоба.

```
static blob Biggles
  private vars x,y,dx,dy
  on_message fly_north with s=speed do
      0 --> dy; s --> dx;
      send 'biggles flies north to base';
  enddo
  ...
  on_message tick with t=time interval do
      x+t*dx --> x;
      y+t*dy --> y;
  enddo
  on_message where_are_you with s=sender do
      send 'biggles is at', x=x, y=y, to (s);
  enddo
endblob
```

Характерный способ реализации системы "классной доски" — создать семейство фреймов, представляющих собой сообщество сотрудничающих экспертов. В каждом фрейме есть слот, содержащий "триггер", и слот, определяющий очередность выполнения процедур, связанных с фреймом. Некоторые фреймы запускают свои процедуры, когда триггер подает сигнал начала процесса. Эти процедуры при прочих равных условиях выполняются в порядке очередности. Очередность может изменяться динамически, если какие-то процедуры записывают факты на классной доске раньше. Для регистрации критических изменений данных на классной доске и исправления соответствующих фреймов можно использовать демоны. Другие фреймы запускаются непосредственно для определенных значений в базе данных.

Объектно-ориентированное программирование — это скорее стиль программирования, чем стратегия представления знаний или управления. Оно было разработано для поддержки модульности и повторного использования кода, а также для усовершенствования представления пользовательского интерфейса. Однако рассматривая системы типа LOOPS компании Xerox, которые сочетают в себе объектно-ориентированное программирование с языком InterLisp, высококачественной графикой и рядом возможностей представления систем, основанных на знаниях, иногда трудно выделить эти свойства. Если известны все атрибуты и методы объекта или блоба, то можно сказать, что мы *понимаем* объект. Наличие свойств и

особенности процедурного наследования приводят нас к пониманию наследования в объектно-ориентированном подходе. Объектно-ориентированные стратегии управления и метауправления определенно имеют интересное будущее. Многое зависит от устранения терминологической неразберихи, так как на данный момент отсутствует четкое разграничение между объектами, фреймами, семантическими сетями и т.д.

10.3.2. Нейронные сети и параллельные вычисления

Сама разработка аппаратных средств тоже попала под влияние объектной философии. Утверждают, что машина AS/400 — объектно-ориентированная в полном смысле этого слова, так как ее аппаратные средства и операционная система разрабатывались с использованием объектно-ориентированных технологий. Операционные системы машин NeXT и Rekursiv также были объектно-ориентированными. С появлением различных моделей параллельных вычислений и соответствующих аппаратных средств объектная ориентация стала естественным способом моделирования обмена сообщениями между элементами компьютерного оборудования. Кроме того, аппаратные средства машины Rekursiv поддерживают передачу сообщений, динамическое связывание и т.д. Параллелизм наблюдается на разных уровнях абстракции — от простых четырехпроцессорных транспьютерных систем до высокопараллельных машин, таких как Connection Machine. Самое высокопараллельное устройство в нашем мире — это человеческий мозг. Нейронные сети — это один из видов высокопараллельных машин, приблизительно повторяющих структуру человеческого мозга. Воспользуемся их примером, чтобы проиллюстрировать, как объектно-ориентированное модельное представление отражается на реализации или моделировании параллельных структур. Но сначала нужно разобраться в том, что такое нейронная сеть.

Теория нейронных сетей начала развиваться со времен появления самих компьютеров, и ее историю можно проследить вплоть до работы Мак-Каллока (McCulloch) и Питтса (Pitts) в 1940-х годах. Как и в случае искусственного интеллекта, когда на первый план вышла обработка коммерческой информации, исследования в этой области переместились в университеты. Тем не менее эта область продолжает оставаться активной и многообещающей. Известные книги Росса Эшби [41, 42] показали, что в 1950-х годах специалисты по кибернетике активно занимались разработкой ряда идей, о которых речь пойдет ниже. Нейрофизиологи также выразили большую заинтересованность, а также сформулировали много интересных идей, которые можно проверить при помощи нейросетевых моделей.

Вероятно, самым значительным из первых примеров практического применения этой идеи можно назвать персептрон Розенблатта (Rosenblatt). Это устройство дает возможность объяснить основные принципы работы нейронных систем.

По сути, идея состоит в том, чтобы создать вычислительное устройство, у которого вместо единого процессора машины Фон Неймана было бы много простых процессоров, наподобие нейронов в мозгу животного. Эти распределенные процессоры, или нейроны, работая вместе, позволяют достичь большего, чем решение самых тривиальных вычислительных задач. Для этого органические нейроны передают сообщения друг другу через огромный клубок соединений. В искусственных нейронных системах между процессорами тоже существуют связи. Персептрон — это простой пример системы, основанной на модели зрительной коры; он спроектирован таким образом, чтобы "учиться видеть" физические объекты, представленные ему (в двухмерном виде) через телекамеру. Распознавание образов — это на самом деле очень сложная задача. Вспомним, как трудно узнать знакомый предмет по фотографии, сделанной под необычным углом зрения или с близкого расстояния. Необходима какая-то система

координат. В случае персептрона и других подобных устройств это достигается введением учителя, который может сказать системе, правильно или неправильно она распознала объект.

Персептрон — это относительно простое устройство, которое можно сравнить с современными нейронными сетями. (С технической точки зрения это — однослойная сеть; подробнее см. ниже.) Мощность современных сетей не уступает мощности цифровых компьютеров (машины Тьюринга). Тем не менее, можно доказать немаловажные теоремы о способности персептрона распознавать образы.

Обычные компьютеры, как правило, состоят из одного центрального процессора и различных периферийных устройств. Информация обрабатывается последовательно согласно заложенной программе. Эта схема обработки информации радикально отличается от структуры человеческого мозга, который, насколько нам известно, состоит из миллионов (приблизительно 10^{11}) очень простых процессоров, называемых нейронами, соединенных между собой еще большим количеством синаптических связей, ведущих к другим нейронам. Чтобы компенсировать простоту нейрона как компьютера и малую скорость передачи информации по синапсам, мозг выполняет обработку информации параллельным способом. Эти открытия в нейронауке вызвали ряд попыток создать вычислительную машину, которая повторяла бы этот параллельный (распределенный) способ обработки информации. Реализации этой идеи сейчас часто называют нейронными сетями.

Связи в нейронной сети позволяют элементам (нейронам) посылать очень простые возбуждающие или подавляющие сигналы. Так как отдельный элемент может получить много (в человеческом мозге — много тысяч) таких сигналов, с каждым входным сигналом связывается "весовой коэффициент", который помогает определить общее влияние сигнала на элемент.

Итак, если более детально, нейронная сеть состоит из

- набора из n процессоров, представляющих собой простые компьютеры, способные вычислить "уровень активации" элемента как функцию входных сигналов и их весовых коэффициентов и вывести значение функции активации;
- вектора n-го порядка, представляющего текущий уровень активации каждого элемента;
- матрицы $n \times n$, определяющей распределение весовых коэффициентов в сети. Отметим, что равенство весового коэффициента нулю означает отсутствие связи;
- правила обучения, по которому веса изменяются с приобретением опыта.

Это слегка упрощенная схема, но заинтересованный читатель может обратиться за более детальной информацией к классической работе [677].

На рис. 10.5 упрощенно иллюстрируется работа фрагмента нейронной сети. Рассмотрим отдельный j-й элемент и вычислим сумму произведений весовых коэффициентов w_{ij} на выходные значения o_i всех нейронов, связанных с данным. Получим значение уровня активации net_j. Затем на основе значения net_j и предыдущего выходного значения нейрона с помощью активационной функции определим выход элемента a_j. Обычно эта функция представляет собой простую сигмоидальную функцию от net_j (рис. 10.6, *а*), которая не зависит от старого значения a_j, но это сильно зависит от задачи. Вычисление выходного сигнала, который распространится по элементам, связанным с элементом j, часто осуществляется при помощи простой пороговой функции (рис. 10.6, *б*). Иными словами, для перехода нейрона в активное состояние входное возбуждение должно достичь определенного уровня.

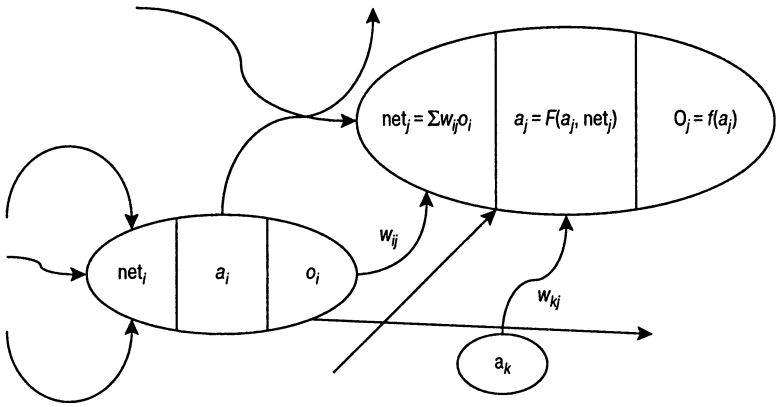
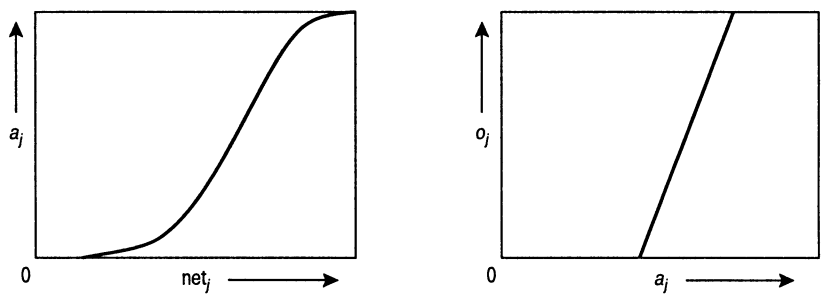


Рис. 10.5. Структура нейронной сети (по Руммельхарту (Rummelhart) и Мак-Клелланду (McClelland))



а) сигмоидальная активационная функция

б) пороговая выходная функция

Рис. 10.6. Типичные функции активации

Алгоритм обучения моделирует способность мозга создавать полезные связи между нейронами, которые обеспечивают распознавание символов или выявление отличительных признаков. Существует множество способов реализации обучения, самый старый из которых — алгоритм обучения Хебба (Hebb), который я опишу в общих чертах.

Правило Хебба усиливает вес связи между двумя активными элементами пропорционально произведению уровня активации второго элемента на выходное значение первого. Таким образом получаем следующее.

$$w_{ij} = ko_i a_j.$$

В более сложной версии этого правила вводится разность эталонного и реального выходов нейрона.

$$w_{ij} = k(t_i - o_i)a_j.$$

Это одно из многих правил обучения, которое известно как Дельта-правило или правило Видроу-Гоффа (Widrow-Hoff). Еще одна популярная идея — рассматривать обучение как процесс минимизации "энергии" или общих затрат сети, которые определяются на основе

суммирования произведений весовых коэффициентов и уровней активации по всем связям. Весовые коэффициенты изменяются в направлении локального минимума, а дополнительные шаги предпринимаются для того, чтобы убедиться, что найден глобальный минимум (что происходит с очень высокой степенью вероятности). Эти методики тесно связаны с генетическими алгоритмами [305], применяемыми в машинном обучении на обычных компьютерах.

Не будем слишком акцентировать внимание на разных методах обучения. По сути, это тема современных исследований, хотя о некоторых алгоритмах известно уже достаточно, чтобы показать, что на их основе можно строить реальные системы.

Элементы нейронной сети удобно делить на скрытые (или внутренние), входные и выходные. Пользователю видны только входные и выходные элементы, а скрытые обычно располагаются во внутренних слоях (по крайней мере, концептуально). Идея состоит в том, что элементы уровня $n+1$ представляют активизацию абстрактных признаков, представленных группой элементов уровня n. Например, мы хотим, чтобы сеть могла распознавать печатные слова. На входные элементы можно подавать буквы из системы оптического распознавания символов. Предположим, существует только один выходной элемент, который активизируется тогда и только тогда, когда входная последовательность — слово. Необходимо предварительно постулировать существование скрытых элементов, которые будут служить детекторами реальных последовательностей. Первый слой отвечает за распознавание букв. Это, конечно, упрощение, и на самом деле необходимо еще включить слой детекторов, определяющих признаки (элементы) букв. Если входные данные соответствуют элементу некоторой буквы, то данный элемент возбуждается, а все остальные — подавляются. Детекторы букв связаны с элементами следующего слоя, которые представляют слова. Если все или некоторые буквы в слове активны, тогда и слово становится активным. Функционирование такой сети определяется алгоритмом обучения (и учителем) или самим разработчиком. Характерный пример такой схемы показан на рис. 10.7.

Нейронные сети хороши как для задач восполнения образов, так и для задач распознавания. Например, в одной из первых исследовательских работ было обнаружено, что некоторые нейронные сети могут научиться распознавать частично зашумленные слова, как показано на рис. 10.8, *а*. Они могут даже научиться делать предположения о "несловах", как показано на рис. 10.8, *б*, выучив некоторые правила о том, что такое "выглядеть как слово". Еще одна немаловажная особенность этих сетей — это способность функционировать даже при повреждении нескольких процессоров. Это свойство называют постепенной деградацией. Если выходит из строя целостная компьютерная система, ее работа прерывается более резко.

Главное наблюдение, которое должно быть сделано, — это то, что нейроны представляют собой объекты с инкапсулированным поведением и общаются при помощи обмена сообщениями. Что может быть лучше языка объектно-ориентированного программирования для моделирования такой системы? В [84] описаны основы объектно-ориентированного каркаса нейронных систем и приводится исходный код библиотеки классов для языка C++, содержащий несколько алгоритмов обучения. Еще одно наблюдение состоит в том, что кратковременная память содержится в самих элементах в виде уровня активации, а долгосрочная память обеспечивается весовыми коэффициентами. Это, мне кажется, приводит к некоторым интересным заключениям о глобальных свойствах объектно-ориентированных систем, где топологию передачи сообщений можно рассматривать как результат инкапсуляции управляющей информации. Эти наблюдения сделаны на основе подходов, описанных в главе 6.

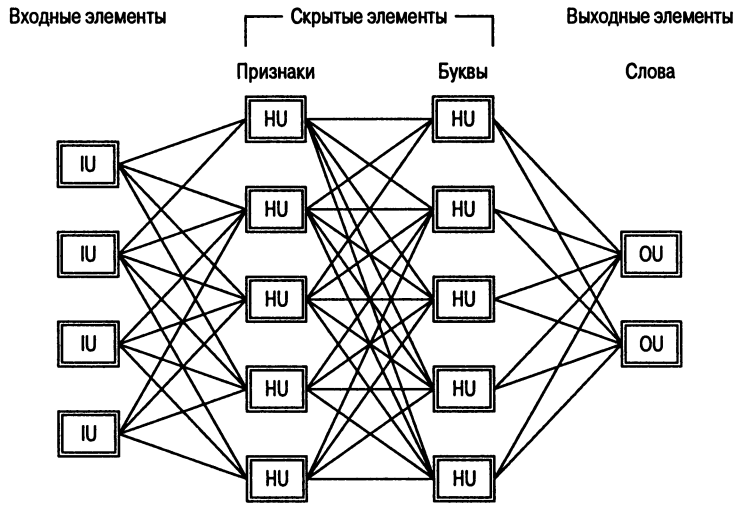


Рис. 10.7. Многослойная нейронная сеть

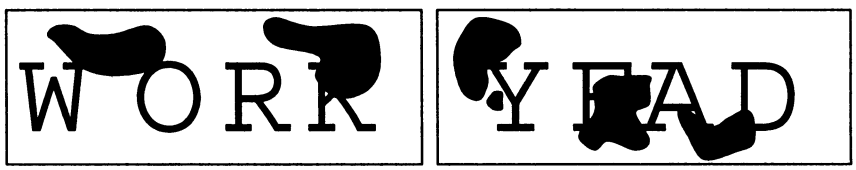


Рис. 10.8. Задача распознавания слова (а) и неслова (б)

10.3.3. Интеллектуальные агенты

Возрастающая сложность коммерческих задач повышает спрос на "мыслящие" программные системы, которые могут работать с конечными пользователями, обдумывая свои задания. Это системы, основанные на интеллектуальных агентах. Некоторые задачи при помощи других технологий решать намного сложнее, например выполнять поиск информации в сети. Агентная технология также обеспечивает эффективное модельное представление для задач анализа, проектирования и реализации. Она облегчает повторное использование на уровне знаний.

Модель интеллектуальных агентов необходимо уяснить по двум совершенно разным причинам. Во-первых, необходимо понять, как моделировать бизнес-процессы с помощью объектного представления агентов, как в главе 7. Во-вторых, следует разобраться, как можно использовать объектно-ориентированные модели для описания компьютерных агентных систем, роль которых все больше возрастает.

Добавление правил к интерфейсам объектов имеет полезный побочный эффект, дающий возможность построить модель интеллектуальных агентов и мультиагентной системы без каких-либо специальных агентных средств моделирования. Агенты — это автономные, гибкие программные объекты, которые могут реагировать на изменения в своем окружении или

контексте, участвовать в "общественных" действиях при помощи общего языка общения и быть в псевдоактивном состоянии, как, например, агенты Intellisense в MS Office.

Одной из главных причин успеха объектной технологии является развитие распределенных архитектур, но n-уровневым архитектурам часто преграждают путь сложные проблемы, связанные с ограниченной пропускной способностью сети. Использование мобильных агентов может уменьшить количество потоков данных в сети. Мобильность агентов означает, что при необходимости, вместо передачи запроса на поиск нефильтрованной информации, можно передать через сеть саму программу. Интеллектуальные агенты можно использовать для "персонализации" систем и их приспособления к потребностям и навыкам пользователя, что приводит к снижению нагрузки на пользователя. Агенты, которые могут изучать, адаптировать и изменять цели и данные, можно применять для ускорения поиска информации, особенно в разного рода сетях.

Еще одна область, где усиливается влияние агентных систем, по крайней мере в качестве модельного представления, — это моделирование и инженерия бизнес-процессов. Процесс, как и положено, находится в центре технологии бизнес-моделирования. Однако, как указано в [751], опасно чрезмерно зацикливаться на процессах, потому что коммерческая деятельность зависит от управления ресурсами и структуры организации, а не только от эффективности процессов. Таким образом, любой подход к моделированию бизнес-процессов должен учитывать все три аспекта: ресурсы, организацию и процессы. Оказывается, агентное представление в комбинации с объектно-ориентированным взглядом на анализ систем дает эффективное решение проблемы моделирования. Более того, моделирование обязанностей пользователя при помощи интеллектуальных агентов часто может снизить диссонанс между тем, как пользователь представляет себе модель системы, и ее действительной структурой.

Что такое агент

Корни агентной технологии исходят из распределенного искусственного интеллекта (DAI — distributed artificial intelligence), хотя популярность этого подхода связана только с появлением другого приложения — мобильных вычислений, разбора почты и поиска информации в сети.

Агенты используются для автоматического мониторинга фондовых бирж и торговли акциями, для нахождения и покупки билетов на недорогие рейсы и сбора информации на компьютере пользователя. Они нашли применение в электронной торговле, где агенты покупателей могут подобрать товары по наилучшей цене среди многочисленных Web-узлов. Агентная система устраняет неполадки на борту космического корабля многоразового использования. Белый Дом использует агенты электронной почты для фильтрации тысяч информационных запросов. В Массачусетсском технологическом институте (MIT) сконструирована агентная система составления расписаний. Исследователи в области робототехники, индустрии развлечений, систем, основанных на знаниях, человеко-машинных систем, баз данных, распределенных систем, коммуникационных сетей, науки о познании, психологии и виртуальной реальности также проявили острую заинтересованность в этой технологии. В число примеров применения агентной технологии на сегодняшний день входят фильтрация и анализ данных, наблюдение за процессами и формирование сигнала тревоги, управление выполнением технологических и бизнес-процессов, доступ к данным/документам и управление их хранением, реализация персональных цифровых помощников, групповая работа с компьютером, построение моделей и компьютерных игр.

Агенты в современных системах выполняют фильтрацию информации, автоматизацию выполнения заданий, распознавание образов и их восполнение, моделирование деятельности пользователей, а также обеспечивают поддержку принятия решений, осуществляют доступ к информации, оптимизацию ресурсов на основе согласования (например, при управлении воздушным движением), составление маршрутов, моделирование и планирование.

В число других современных областей применения входят:

- агенты пользовательского интерфейса (такие, как Office Assistant от компании Microsoft);
- командование и управление на поле боя;
- мониторинг процессов (в сетях и бизнес-процессах).

Вместе с изобилием новых областей применения появилось большое количество новой запутанной терминологии, с чем сталкивается всякий, кто хочет разобраться в принципах работы интеллектуальных агентов. Встречается много противоречивых и перекрывающихся терминов, таких как интеллектуальный агент (intelligent agent), база знаний (Knowbot, Softbot или Taskbot), персональный помощник (personal assistant) и мастер (Wizard). Также часто встречаются описания сетевых агентов, которые в действительности не являются тем, что подразумевается под большинством из приведенных выше названий. Более того, в литературе существует несколько конкурирующих определений интеллектуального агента. В [679] агент характеризуется как "объект, воспринимающий свое окружение при помощи сенсоров и воздействующий на него при помощи эффекторов. Интеллектуальный агент — это тот, который действует правильно". Доклад [346] несколько ужесточает это определение: "Агент — это самодостаточный элемент программного обеспечения, отвечающий за исполнительную часть программного процесса, обычно в распределенной среде разработки. Интеллектуальный агент пользуется непроцедурной информацией о процессе — знаниями, которые определены в базе знаний и доступны с помощью механизмов логического вывода". Но более точное определение дано в [660]: "Объект является программным агентом тогда и только тогда, когда он правильно общается с подобными себе путем обмена сообщениями на *языке общения агентов*". Это самое важное условие взаимодействия агентов разных производителей. В [439] сказано, что агенты — это объекты, "активно реализующие автономное поведение и взаимодействующие путем согласования".

Языки общения агентов (agent communication language — ACL) выполняют роль, очень напоминающую роль брокеров объектных запросов, и могут применяться поверх них. Такие языки необходимы для того, чтобы агенты можно было рассматривать как распределенные объекты, которым при создании совсем не обязательно знать о существовании друг друга. Существует два вида языков общения агентов — процедурные, такие как язык Telescript от компании General Magic, и декларативные, такие как KQML/KIF Европейского космического агентства.

Можно добавить, что агент — это объект, который может чувствовать, принимать решения, действовать, общаться с другими объектами, перемещаться, сохранять свои представления и учиться. Не каждый агент обладает всеми этими способностями, но нужно, по крайней мере, их учитывать. Один из способов сделать это — классификация агентов по уровню возможностей, которыми они обладают. Рассмотрим четыре уровня агентов в порядке возрастания сложности и мощности.

- Базовые программные "агенты"
- Реактивные интеллектуальные агенты
- Совещательные интеллектуальные агенты
- Гибридные интеллектуальные агенты

Зачастую термин "агент" применяется для описания обычных модулей кода, которые выполняют заранее определенные задачи. Особенно распространено его употребление по отношению к макросам, связанным с таблицами, или триггерам и хранимым процедурам баз данных. Такие "агенты" обычно занимают отдельное место и не могут ни обучаться, ни адаптироваться, ни действовать коллективно. Они также явно не управляемы. Это название также применяют к простым почтовым агентам и Web-макросам, написанным на языке PERL (Practical Extraction and Report Language) или TCL (Tool command language) [611]. Еще более простой пример — сценарии автоматизации работы персонального компьютера, которые можно найти в среде разработки New Wave корпорации HP.

Реактивные интеллектуальные агенты представляют собой категорию самых простых агентов с абсолютно соответствующим названием. Это программы, управляемые данными; и это означает, что они реагируют на стимулы и не имеют целевой ориентации. Они выполняют заранее определенные задачи, но могут реализовывать и символьные логические рассуждения, зачастую основанные на правилах. Иногда они могут общаться с другими агентами. Они могут обладать способностью к обучению и развитию интеллекта. На макроуровне они иногда могут обеспечивать явное управление, а на микроуровне — неявное. Они не могут рассуждать об организации. Обычно применяются однородные группирования таких агентов. Примеры реактивных интеллектуальных агентов — агенты мониторинга и предупреждения, представленные в виде набора источников знаний или правил глобальной стратегии управления.

Совещательные интеллектуальные агенты — это, главным образом, программы с целевым управлением. Они могут устанавливать новые цели и следовать им. При их реализации, как правило, используются символьное представление и рассуждения на основе продукционных правил. Они обычно поддерживают модель своих представлений об окружающей среде и состояние процесса поиска цели. Они могут быть мобильными и общаться, т.е. обмениваться информацией (или даже целями) с другими агентами, которых они встречают. Совещательные агенты могут обладать способностью к обучению. Они могут рассуждать об организации и способны на сложные логические построения. Их интеллект запрограммирован на микроуровне, на котором возможно явное управление. Допустимы однородные группы таких агентов. Агенты доступа к данным, которые извлекают и фильтруют данные из базы данных или из сети, — характерные примеры этого вида агентов.

В некоторых источниках [439] описаны **слабые агенты** — автономные, мобильные, реагирующие на события, способные влиять на свое окружение и взаимодействовать с другими агентами. **Сильные агенты** дополнительно обладают свойствами хранения представлений, целей и планов действий, обучения и достоверности. Хотя важность последнего свойства активно обсуждается.

Гибридные интеллектуальные агенты — это комбинация совещательных и реактивных агентов. Такие агенты могут быть мобильными и могут активно и динамически сотрудничать с другими агентами. Они также обучаемы, это — **сильные гибридные интеллектуальные агенты**. Такие агенты обычно содержат базу знаний правил и утверждений (представлений) (или имеют доступ к ней) и библиотеку планов. Интерпретатор дает возможность агенту выбрать

план, соответствующий его текущему состоянию и целям. Когда происходит событие, выбирается (путем присваивания значения) план, отражающий текущее состояние агента.

Ловушки

Несмотря на огромные потенциальные преимущества, технология интеллектуальных агентов все еще содержит множество открытых вопросов. Теории агентов еще далеки от завершения. До сих пор отсутствуют методологические руководства. Выбор средств разработки все еще ограничен, что уже обсуждалось выше. Базовая технология потенциально очень сложна. И наконец, еще нужно собрать большое количество практической и аналитической информации из реальной жизни и крупномасштабных приложений.

Разработка агентов остается чем-то из области черной магии, потому что нет ни одного опубликованного пособия по выбору нужной архитектуры агентов для решения различных классов задач. Разработчик также должен постараться найти ответы на вопросы о том, как агент приобретает свои знания и как эти знания могут быть представлены. Нужно решить, как выполнить декомпозицию сложных задач и распределить их решение по разным агентам. Следует подчеркнуть и проблему "языка общения агентов": как однородные агенты могут сотрудничать и общаться? И наконец, появляются новые проблемы законности и этики: всегда ли можно доверять разумному агенту? Человек может решить, что его работа находится под угрозой, и это также может послужить препятствием для принятия технологии.

Выше было показано, что существует очень много определений агентов и подходов к ним, многие из которых кажутся сомнительными. Ожидается, что сегодняшняя тенденция к наделению компьютеров большим интеллектом будет развиваться, но темпы развития могут быть замедлены отсутствием квалифицированных источников. В ближайшее время, по-видимому, мы сможем наблюдать дальнейшее распространение агентного программного обеспечения в область мобильного программирования и глобальных сетевых вычислений. Однако остается множество проблем, как теоретических, так и практических. Можно ожидать, что использование объектно-ориентированных методов и средств для моделирования и реализации агентных систем возрастет. Ясно, что расширится и использование распределенных объектно-ориентированных технологий, таких как технология CORBA, не пользующихся агентами. С другой стороны, интеллектуальные агенты — это очень многообещающая технология, способная помочь осуществить крупномасштабные объектно-ориентированные разработки. Появление стандартных агентных архитектур и языков общения агентов может привести к появлению рынка коммерческих агентов.

Архитектура агентных систем

Интеллектуальные агенты обладают разумом в том смысле, что имеют определенный опыт или способность к обучению. Эти знания могут быть представлены в виде продукционных правил, тогда агент должен быть оснащен механизмом логического вывода для их выполнения. Алгоритмы обучения обычно функциональны по своей природе и могут основываться на построении вариантов решений (например, алгоритм ID3), нейронных сетях или генетических алгоритмах. Агентам необходимо общаться с пользователями и друг с другом. Как я уже говорил, лучше всего для этого использовать стандартный язык общения агентов. К несчастью, такой универсальный стандарт еще не утвержден, так что разработчики часто вынуждены его создавать или работать в рамках собственной среды.

Можно построить базовую архитектуру, характерную для большинства агентных систем. Такая архитектура показана на рис. 10.9. У каждого агента есть контроллер, который содержит механизм логического вывода и стратегии решения задач или обеспечивает доступ к ним. Агент в своей базе инкапсулирует два вида знаний: постоянные и переменные. Постоянные знания обычно принимают форму атрибутов и методов, которые представляют навыки агента в объектно-ориентированной реализации, но методы могут быть представлены непроцедурно, например с помощью общепринятого языка реализации Prolog. Агент также может содержать знания о тех ролях, которые он исполняет в общей структуре агентов, и о своих планах. Фиксированные планы следует отличать от тех, которые изменяются во время выполнения. Последние являются частью базы переменных знаний агента и включают его текущие предположения, представления, связи и кратковременные цели. Агенты общаются через очередь сообщений и отправляют сообщения в "почтовое отделение" сети, которое передает их другим агентам, системам или пользователям. Читатель может заметить сходство этой структуры общения со структурой брокеров объектных запросов, значит, последние можно использовать для реализации агентно-ориентированных систем. Реальные агентные системы сильно отличаются друг от друга, но по крайней мере в общих чертах соответствуют этому основному подходу.

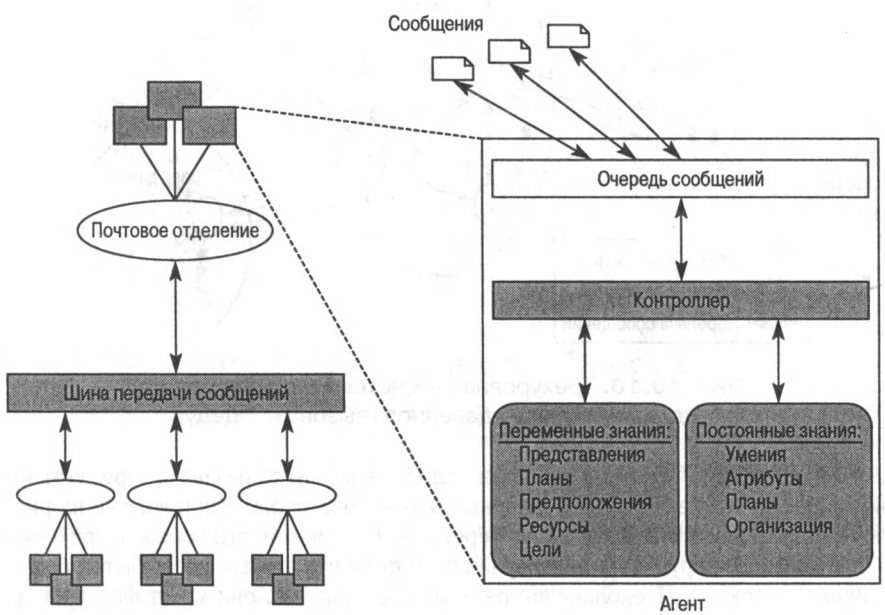


Рис. 10.9. Типичная архитектура агента

Одна из главных проблем, связанных с современными системами клиент/сервер, — это объем сетевого трафика. Даже в высокоскоростных локальных сетях это может привести к непредсказуемому снижению производительности, что для многих хуже, чем прогнозируемое медленное выполнение. Многие компании попытались решить эту проблему путем перехода от двухуровневых систем первого поколения со всеми их дополнительными проблемами, связанными с вызовами хранимых процедур, к трехуровневым системам, основанным на вызове

удаленных процедур (RPC — remote procedure call). В трехуровневых системах большая часть логики процесса перемещается на промежуточный "сервер приложений", который доступен для клиентов, нуждающихся в услугах приложений. В настоящее время доступ чаще всего обеспечивает брокер объектных запросов. Обращения к базе данных в настоящее время чаще всего реализуются на языке структурированных запросов SQL (Structured Query Language). Типичная архитектура такой системы показана на рис. 10.10. Заметьте, насколько при этом может быть загружена сеть. По большей части это происходит потому, что приложения осуществляют доступ к таблицам, содержащим большое количество информации, не интересующей пользователя. Также следует обратить внимание на необходимость использования контроллеров. Это объекты, способные управлять многими потоками выполнения и поэтому нуждающиеся в доступе к реализациям других объектов или, в самом крайнем случае, в сложных знаниях о них. Таким образом, существует опасность, что нарушение инкапсуляции может перечеркнуть все преимущества использования объектной ориентации.

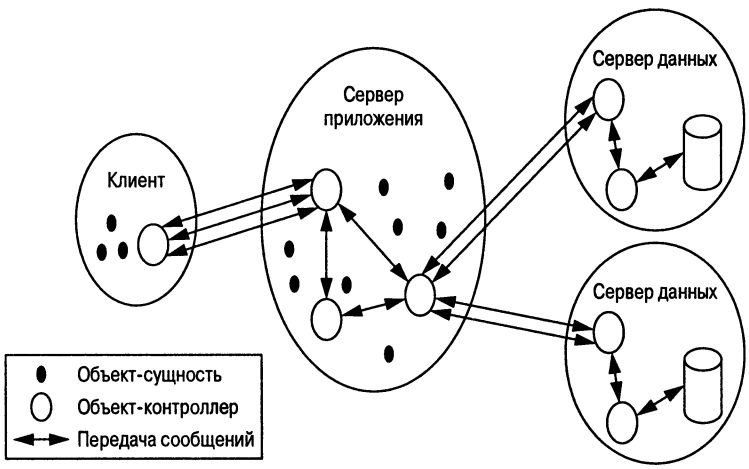


Рис. 10.10. Трехуровневая система клиент/сервер: подход на основе удаленного вызова процедур

На помощь приходят агенты. При помощи агентов можно передавать через сеть программу, а не запрос на языке SQL или удаленный вызов процедуры. Как показано на рис. 10.11, мобильный агент (или агент-посланник) берет на себя обязанности контроллера и может это делать без ужасного нарушения инкапсуляции (потому что агенты могут быть представлены как истинные объекты). Поскольку агенты обладают логикой, они могут фильтровать полученную информацию и возвращаться к клиенту с пакетом только интересующих его данных. Таким образом уменьшается количество информационных потоков для целого ряда приложений, в которых поиск осуществляется по сложным критериям. Это размывает границу между клиентом и сервером, приближая агентные системы к архитектуре взаимодействия равноправных узлов в локальной сети ("точка-точка"). Более того, решение о размещении кода может быть принято на более поздних этапах проектирования, что уменьшает риск принятия неправильных решений.

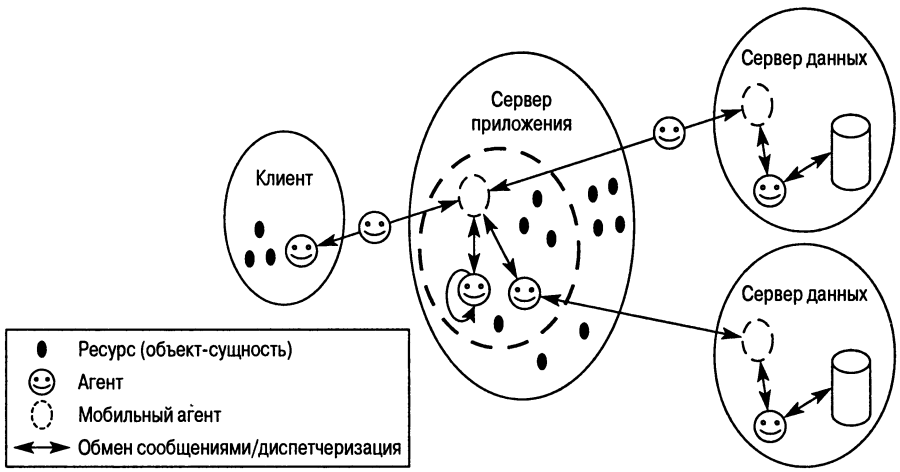


Рис. 10.11. Трехуровневая система клиент/сервер, построенная с использованием агентов

Агентная модель — это модель распределенного решения задачи. Существует несколько подходов к совместной работе распределенных агентов. Это централизованное управление, модели соглашений (как при объектно-ориентированном подходе или в примере сети Contract Net Смитта (Smith)), иерархическое управление через организационные элементы, мультиагентные системы планирования и модели согласования. Мы не будем их здесь детально обсуждать, но один тип стратегии особенно важен для систем, содержащих множество взаимодействующих агентов, каждый из которых применяет специализированные знания для решения общей задачи. Общепринятая архитектура таких приложений — это "неявное обращение" или модель "классной доски".

Моделирование агентов с помощью объектов

Вернемся к обсуждению взаимосвязи между агентами и объектами и покажем, как соответствующее расширение объектно-ориентированных методов можно использовать для моделирования агентов. Это органически приводит к вопросу использования агентного представления для моделирования бизнес-процессов.

Разработчикам агентных систем нужна технология моделирования, и возникает естественное желание использовать для этой цели объектную технологию, потому что агенты, как и объекты, очевидно инкапсулируют и состояние, и поведение. Конечно, для построения агентных систем можно использовать объектно-ориентированные языки, такие как Java, CLOS и т.д., но большинство объектно-ориентированных методов на самом деле не позволяет представлять интеллектуальные агенты, инкапсулирующие способность логического мышления. В действительности в большинстве агентных реализаций используется смесь процедурных и непроцедурных языков, например языка C с языком Prolog. Трудность моделирования состоит в том, что большая часть этих методов не позволяет объектам инкапсулировать информацию второго порядка о самих себе: о связях между атрибутами и операциями. Несколько продвинулись в решении этой проблемы методы, поддерживающие инварианты класса, например BON (bed-of-nails) [772], но с их помощью все еще трудно представлять

способность интеллектуальных агентов делать логические выводы. Инварианты метода BON должны быть записаны при помощи процедурного кода на языке Eiffel. Однако при этом теряется непроцедурный характер продукционных правил. Такие методы не могут удовлетворить потребность в построении цепочек правил и осуществлении логического вывода (поиска) на их основе. Единственный из известных автору методов, инкапсулирующих наборы правил внутри объекта, — это метод SOMA.

Следовательно, любая попытка создать метод для агентных систем на основе объектно-ориентированных методов первого поколения обречена на поражение. Мне довелось наблюдать несколько попыток использования смеси методов OMT, Objectory, RDD и т.п. В каждом из этих случаев возникали проблемы. Сравнивая понятия, принятые в агентных системах, с терминологией, которой пользуются создатели объектно-ориентированных методов, следует отметить, что агенты осуществляют мониторинг событий в базах данных и выполняют триггерные операции, разнородным агентам необходимы онтологии для сотрудничества, интеллектуальные агенты содержат правила и могут рассуждать логически, а некоторым интеллектуальным агентам приходится делать логические выводы в условиях неопределенности. Объектно-ориентированные методы обычно не поддерживают этих операций. К тому же агенты управляют собственными потоками управления. Эти операции тоже поддерживаются очень слабо (за исключением систем исполнителей).

Проблемой является также моделирование координации агентов, распределения заданий и знаний, а также взаимодействия целей, где агентам могут потребоваться рассуждения о представлениях и стратегиях координации других агентов. Заметим, что некоторые требования к агентным системам моделирования не поддерживаются совсем или поддерживаются в рамках нестандартных подходов с использованием делегирования обязанностей и активных объектов. Однако такой поддержки может быть недостаточно для агентов, и она может не соответствовать объектно-ориентированным принципам. Можно отметить, что для управления потоками применяют объекты-контроллеры Якобсона (Jacobson). Однако при их реализации зачастую нарушается инкапсуляция управляемых объектов. При разработке агентов инкапсуляция должна сохраняться, чтобы они могли выполнять функцию управления другими объектами. Это легче всего осуществить в рамках пропагандируемого здесь подхода, потому что агенты общаются только с интерфейсами других объектов и другими агентами (используя преимущественно язык общения агентов). Это помогает избежать опасности возникновения очень сложных проблем поддержки, когда "управляющие" объекты нарушают инкапсуляцию.

Функциональный элемент FUN (functional unit) — понятие, введенное в [266] для выражения идеи составного агента: множество агентов, объединенных в одну группу внутри определенной организационной структуры и инкапсулированных в рамках одного интерфейса. Интерфейс перехватывает все сообщения и делегирует их соответствующим агентам из команды. Такая структура в методе SOMA называется **оболочкой** (wrapper). Функциональные элементы могут использоваться для моделирования ролей, которые может выполнять агент. Функциональный контекст может изменить предположения о поведении агента и таким образом снизить уровень общения. Отдельные агенты выполняют задачи, которые могут быть представлены как операции в их интерфейсах. Эти задачи могут быть извлечены из описания класса, к которому принадлежит агент, и сохранены в самих ролях. Роли похожи на классы, за исключением того, что их экземпляры могут переходить в другие роли (динамическая классификация). Иными словами, агенты отвечают за выполнение задач, описываемых ролями. Кроме того, агент может быть определен как компонент функционального элемента, содержащий информацию о ролях и обеспечивающий соответствующие подстановки при выходе агента из строя.

Как оказалось, большая часть указанных выше проблем решается при помощи метода SOMA, даже несмотря на то что он был разработан не для этой цели, а для расширения объектного подхода на задачи моделирования бизнес-процессов. Я попытаюсь объяснить это замечательное совпадение.

Должно быть ясно, что наборы правил — это все, что необходимо для превращения объекта в агент. Прямые цепочки наборов правил (управляемые данными) обеспечивают возможность создания реактивных агентов, а совещательные агенты поддерживаются в режиме обратной цепочки. Так как каждый объект может содержать несколько наборов правил, а каждому набору правил может соответствовать отдельный режим, то в рамках этого подхода можно описывать и гибридные агенты. Отметим, что способность к обучению обычно нельзя представить в виде набора правил. Более подходящим представлением являются операции или смесь того и другого.

Архивы бизнес-объектов составляют, по сути, онтологию данной предметной области. Эта объектно-ориентированная онтология расширяет концепцию словаря данных не только за счет включения поведения в форме операций, но и за счет инкапсуляции неявных бизнес-правил.

Интеллектуальные агенты должны работать в условиях различной неопределенности: вероятностной, неопределенности возможностей и т.д. Системы моделирования, в которых правила или инварианты класса выражаются в форме стандартной логики предикатов или исчисления предикатов первого порядка (FOPS — first order predicate calculus), слишком ограничены. Метод SOMA позволяет разработчикам выбирать логику рассуждений: FOPS, временную логику, нечеткую, этическую (логику обязательств или долга) и т.п. Набору правил соответствует свой режим логического вывода и своя логика. Фактически режим и логика тесно связаны. Например, стандартная нечеткая логика подразумевает (обычно) одношаговую стратегию построения прямой цепочки и параллельную обработку правил. В методе SOMA поддержка нечеткой логики особенно сильна: допускаются нечеткие объекты с нечеткими атрибутами и частичным (нечетким) наследованием.

Объекты с наборами правил обеспечивают моделирование и проектирование интеллектуальных агентов и систем, но агенты, в свою очередь, являются ключом к моделированию бизнес-процессов и снижают познавательный диссонанс между моделями мира и разработкой систем.

Моделирование бизнес-процессов при помощи агентов

До сих пор речь шла о технологии интеллектуальных агентов и о способе моделирования агентных систем методами объектно-ориентированного анализа. Предполагалось, что для разработки компьютерной системы на базе мобильных интеллектуальных агентов на этапе формулировки требований и в процессе проектирования системы можно воспользоваться методом SOMA. Итак, наше обсуждение агентной парадигмы поднимается с уровня кодирования на более абстрактный уровень анализа, освобождая при этом модель от языковых и архитектурных зависимостей. Теперь же я хочу перевернуть все с ног на голову и спросить, можно ли применять агентную парадигму для описания систем, существующих в реальном мире, а не внутри машины? Метод SOMA рассматривает интеллектуальные агенты в двух ракурсах:

- как метод реализации;
- как модель бизнес-процесса.

Именно на последний пункт мы и обратим особое внимание.

Объектная модель агента, которая является и моделью бизнес-процесса, определяет каркас, в рамках которого происходит вся деятельность. Изначально она обеспечивает описание участников, вовлеченных в разные виды деятельности и правила их взаимодействия (переговоры). Как следует из главы 8, эти участники являются агентами. Это обеспечивает высокий уровень рассмотрения процесса разработки системы. Чего агентная модель не может обеспечить в явном виде — это ввести какое-либо понятие *последовательности*. Следующий уровень детализации последовательности действий обеспечивается с использованием диаграмм последовательностей или множеств ассоциации задач.

Как уже говорилось, внешние агенты напоминают "черные ящики", находящиеся вне нашего контроля и организации. Внутренние агенты работают в рамках контекста, и их мотивы нам известны. И те, и другие могут выполнять роли непосредственных пользователей системы и становиться таким образом исполнителями. Однако для внешних агентов роль пользователя необычна. Если же это все-таки случается, то внешний агент выступает в роли исполнителя. Внутренние и внешние агенты и исполнители общаются при помощи обмена сообщениями. Сообщение появляется в результате события.

В заключение можно констатировать следующее. Во-первых, обогащенные правилами объектно-ориентированные методы, такие как метод SOMA, очень удобны для моделирования агентных систем, а семантически менее богатые методы не подходят для этой цели или создают ненужные сложности. Во-вторых, использование агентной парадигмы на начальном этапе описания требований или обратного проектирования бизнес-процессов обеспечивает теоретически верный путь определения требований пользователей и их взаимосвязи с задачами производства. Эти требования можно материализовать в виде объектов-заданий с большей адекватностью, чем без использования объектно-ориентированной технологии, в том числе на основе прецедентов.

10.4. Назад в будущее

В предыдущих изданиях этой книги я сделал ряд прогнозов о направлениях развития информационных технологий. Сейчас объектно-ориентированные методы более устойчивы и общепринятыи и можно оценить, насколько точными были прогнозы.

Языковые тенденции

Я предсказывал, что большая часть языков третьего поколения приобретет объектно-ориентированные особенности, как это уже произошло с языками C, Lisp, COBOL и Pascal. Я пошел немного дальше и предсказал, что эти расширения станут общепринятыми и объектно-ориентированные свойства будут реализованы на уровне базового компилятора и станут частью международных языковых стандартов. Даже язык COBOL не смог противостоять этой тенденции. Язык C был на удивление стоек перед лицом давления, оказываемого путем добавления классов. А язык Object COBOL, кажется, не "просоответствовал" основным тенденциям ИТ. Основное требование здесь — не написание нового кода, пригодного для употребления, а повторное использование миллионов строк существующего кода языка COBOL и его расширение или постепенная замена. Языки объектно-ориентированного программирования на сегодняшний день проникли даже в самые консервативные организации, занимающиеся обработкой данных.

Парадоксально, но я не верил в перспективы языка Object COBOL, который во всем опирается на язык Ada, и это подтвердилось фактом снижения интереса к нему. Как и предсказывалось, язык C++ остается наиболее предпочтительным для разработок, в которых недопустимы ограничения языка Smalltalk, касающиеся производительности. Конечно, в 1994 году я не знал, что во многих основных приложениях языки Smalltalk и Eiffel заменит Java. Я говорил, что для задач, в которых важна декларативная семантика, лучше всего подойдет язык CLOS. Известно так мало подтверждений этой гипотезы, которые привлекли бы внимание общественности, что этот прогноз также можно считать неоправдавшимся. Встроенный искусственный интеллект действительно существует, но о языке CLOS в научных кругах услышишь нечасто.

В число открытых исследовательских задач входили и по-прежнему входят объединение функционального и объектно-ориентированного стилей; взаимосвязь между объектно-ориентированным, логическим программированием и формальными методами; эффективность; разработка теории типов для языков объектно-ориентированного программирования; и возможность смешанной языковой среды, обеспечиваемой небольшими интерфейсами между языками, которые можно рассматривать как инкапсулированные объекты. Последняя проблема лишь частично решается с помощью программного обеспечения среднего уровня, что обсуждалось в главе 4. Введение механизма агентов в язык Eiffel подтверждает стремление к объединению функционального и объектно-ориентированного программирования. Все еще существует ряд открытых вопросов, связанных с объектно-ориентированными базами данных, включая вопросы эволюции схемы, стратегий блокировки, распределенного доступа и языков запросов. Я говорил, что реляционные базы данных будут приобретать объектно-ориентированные черты и уже через десять лет трудно будет заметить разницу. Не премину заметить, что по большей части это уже произошло на четыре года раньше, как мы видели в главе 5.

Я утверждал, что со снижением затрат на компьютерное оборудование усилится стремление к "сборке мусора" и динамическому связыванию. Язык Java подтвердил это утверждение, но сейчас Java еще далек от того, чтобы считаться последним словом среди языков объектно-ориентированного программирования. Хорошо видна перспектива совершенно новых языков, объединяющих в себе уроки как языка Eiffel, так и CBD. Я, может быть, переоценил значение параллельного программного обеспечения, но, по крайней мере, многопроцессорность сегодня стала обычным явлением.

Языки четвертого поколения, как и предсказывалось, продолжают приобретать объектно-ориентированные черты, но стали менее значимыми, я надеюсь временно, для основных направлений разработки. На мой взгляд, программное обеспечение экспертных систем все больше сближается с еще не реализованными языками четвертого поколения. Идеи экспертных систем в объектно-ориентированном анализе, которые обсуждались в этой книге, еще должны повлиять на разработку новых языков. Фактически снижение интереса к языку Eiffel — это шаг назад. Я все еще уверен, что будущее будет немного другим. С другой стороны, мое предположение, что эволюционная разработка вскоре станет нормой в программной инженерии, почти полностью подтвердилось.

Я убеждал, что технология репозитариев очень напоминает объектно-ориентированные базы данных и с уверенностью можно ожидать, что в течение нескольких лет все CASE-архивы будут объектно-ориентированными. Появление объектно-реляционных баз данных одновременно и подтвердило, и опровергло этот прогноз. Объектно-ориентированные базы данных не заменили реляционные для маленьких систем и для систем, содержащих сложные типы данных, за исключением нескольких ключевых областей — так что я опять оказался

неправ. Я также говорил: "реляционные СУБД останутся средой разработки для большинства коммерческих проектов, но количество объектно-ориентированных свойств увеличится". Так и случилось.

Объекты, искусственный интеллект и неопределенность

Я комментировал расхождение в терминологии специалистов по искусственному интеллекту, базам данных и объектно-ориентированным методам. Фреймы в первых, сущности во вторых и объекты в последних так мало отличаются друг от друга, что я предсказывал полное слияние этих понятий, предлагая в качестве одного из вариантов общего понятия объекты метода SOMA с их наборами правил. Принципиальными пунктами при слиянии этих понятий должны были быть следующие.

- Фреймы искусственного интеллекта должны более четко инкапсулировать методы и обеспечивать более высокую степень сокрытия информации.
- Фреймы должны поддерживать множественное наследование неявно инкапсулированных методов, а также значений атрибутов.
- Объекты должны инкапсулировать декларативную семантику и правила управления.
- Объекты должны обладать способностью инкапсулировать методы для области определения своих атрибутов в форме демонов, которые запускаются, кода нужно установить значение атрибута или когда атрибут обновляется.
- Объектно-ориентированные системы должны допускать множественное наследование значений атрибутов, а также методов.
- Объектно-ориентированные системы должны сочетать средства моделирования и управления неопределенностью.

У нас еще есть путь к отступлению. С течением времени коммерческий интерес может сфокусироваться на чем-то еще, но я остаюсь при убеждении, что для этого необходим длительный период. Нечеткие объекты также нашли ограниченное применение только в таких областях, как ГИС. Я уже писал и повторю ниже, что управление неопределенностью не изменилось до сегодняшнего дня.

В объектно-ориентированной литературе всегда очень мало внимания уделялось проблемам управления неопределенностью. Однако давно известно, что управление неопределенностью — это ключевая проблема в принятии решений и задачах управления процессами. Те, кто принимают решения, хотят получать ответы быстро; а их точность не является главным требованием. Кроме того, может быть принято несколько приблизительно правильных решений, а входные данные могут быть слишком неопределенными или дорогими, чтобы подбирать их с абсолютной точностью. Более того, известно, что теория вероятности не в состоянии описать все виды неопределенности. В [70] об этом сказано так.

> "Большая часть процессов принятия решений в реальном мире происходит в среде, где цели, ограничения и последствия возможных действий точно не известны. Чтобы количественно оперировать неточностью, мы обычно применяем принципы и методики теории вероятности и, в отдельных случаях, методы теории принятия решений, теории управления и информации. Поступая так, мы неявно принимаем предположение о том, что неточность —

какова бы ни была ее природа — можно приравнять к случайности. Это, с нашей точки зрения, спорное утверждение.

Говоря точнее, противоречие состоит в том, что существует необходимость различать *случайность* и *неопределенность*, так как последняя выступает главным источником неточности во многих процессах принятия решений. Под неопределенностью мы понимаем тип неточности, который связан с *нечеткими множествами*, т.е. классами, в которых нет резкой границы между членами и нечленами. Например, класс *зеленых объектов* — это нечеткое множество. Такими являются классы объектов, характеризуемых такими общими прилагательными, как большой, маленький, материальный, значительный, важный, серьезный, простой, точный, приблизительный и т.д. Действительно, в отличие от понятия класса или множества в математике, большинство классов в реальном мире не имеет жестких границ, которые отделяют те объекты, которые принадлежат данному классу, от тех, которые ему не принадлежат. В этой связи немаловажно заметить, что в человеческих разговорах неопределенные утверждения типа "Джон на *несколько* дюймов выше, чем Джим", "*X намного больше*, чем *Y*", "корпорация *X* имеет *большое будущее*", "фондовая биржа претерпела *сильное* снижение цен" несут информационную нагрузку, несмотря на неточное значение выделенных слов. Фактически можно доказать, что главное различие между человеческим разумом и машинным интеллектом состоит в способности человека (которой не обладают сегодняшние компьютеры) оперировать неточными понятиями и выполнять нечеткие указания."

Большинство методов управления неопределенностью берет свое начало из работ по исследованию операций и экспертным системам. Методы, которые использовались или предлагались всерьез, делятся на два широких класса: те, которые предлагают некий способ количественного численного представления степеней определенности, и те, которые пытаются моделировать неопределенность чисто описательным или качественным способом.

Численные методы включают теорию нечетких множеств, описанную в общих чертах в приложении А, факторы уверенности и байесовскую теорию вероятности. В модели медицинской диагностической экспертной системы MYCIN, на основе фактора уверенности и ее последующих расширениях, каждому утверждению присваивается коэффициент уверенности от -5 до +5 и для распространения этих факторов по длинным цепочкам логических рассуждений используют вычисления, математически схожие со стандартными вычислениями теории нечетких множеств. Сами правила также могут быть неточными, и это принимается во внимание. Обобщением этого подхода с более развитым математическим аппаратом является теория доказательств Демпстера-Шафера (Dempster-Shafer). Еще одна экспертная система, PROSPECTOR, основана на байесовской теории вероятности. Здесь уверенность в высказывании интерпретируется как субъективная вероятность и передается в соответствии с правилом Байеса. В главе 2 книги [331] эти методы описаны в ознакомительном стиле.

Таким образом, существует потребность представления неопределенных утверждений в объектно-ориентированных языках программирования. В настоящее время все эти функции приходится выполнять вручную, так как ни один язык объектно-ориентированного программирования не имеет подходящих встроенных конструкций. Экспертные системы-оболочки обычно не намного лучше, так как они не предоставляют никаких способов описания нестрогой информации, в лучшем случае всего один или два.

Один из немногих коммерческих продуктов в этой области, который предлагает управление неопределенностью, основанное на нечетких множествах, — это система ES/KERNEL, представляющая собой объектно-ориентированную экспертную систему-оболочку, поставляемую на рынок в Японии фирмой Hitachi. Она использовалась для решения ряда задач.

Данная система обладает всеми обычными возможностями оболочки, но еще имеет и ряд новых особенностей. Она поддерживает представление знаний в виде правил и объектов в объектно-ориентированном стиле, у нее отличный графический пользовательский и программный интерфейс — на японском языке — и, что здесь особенно замечательно, она содержит возможность неточного вывода, представленную как расширение основного продукта. Эти нечеткие средства управления очень хорошо реализованы. Рассмотрим их очень кратко.

Функции принадлежности (нечетких множеств) выбираются из семи возможных графических образцов, которые пользователь может корректировать, изменяя два, а иногда и больше параметров. Эти параметры могут изменяться динамически в процессе рассуждений. Нечеткая часть системы ES/KERNEL берет свое начало от удачной нечеткой экспертной системы управления, разработанной для метрополитена города Сендай (Sendai), о которой речь пойдет ниже. Этот пример практического применения служит основой для уверенности в общности семи форм функций принадлежности, но, возможно, ограничивает диапазон возможных применений системы в ее теперешнем виде. Ниже приведен характерный пример синтаксиса нечеткого правила (в финансовом приложении).

```
ЕСЛИ официальный курс высокий и средняя рыночная стоимость низка
ТО покупать меньше фунтов
ЕСЛИ официальный курс низкий и средняя рыночная стоимость высока
ТО покупать больше фунтов
```

Я выделил курсивом фразы, соответствующие функциям принадлежности. Механизм неточного логического вывода — это надстройка к базовой системе ES/KERNEL. Связь между базой нечетких правил и ядром системы осуществляется путем обмена сообщениями между правилами или процедурами (методами) и фреймами вызова нечеткой системы. Ее методы рассуждений особенно удобны для применения в тех областях, где требуется управление в реальном времени и прогнозирование.

Одним из самых впечатляющих приложений нечетких экспертных систем является система автоматического управления поездами, разработанная фирмой Hitachi для Сендайской городской системы метрополитена. Немаловажно, что пассажиры Сендая готовы доверить свою безопасность математическим расчетам, которые часто подвергаются несправедливой критике. Преимущества, которые они при этом получают, — это, главным образом, более равномерное движение и низкие цены на проезд, так как контроллер также оптимизирует расход топлива и другие аспекты работы системы в целом. Ключ к успеху нечеткого управления, в противоположность численному, — это учет правил работы опытных машинистов. Эти правила, прежде всего, выражают неопределенные условия: "если скорость *намного ниже* предельной, тогда повышается уровень мощности". Кроме того, такие показатели системы, как безопасность и комфорт, заранее выражаются функциями принадлежности. Тогда стандартные методы нечетких множеств можно использовать для учета гибких ограничений в управляющем воздействии за счет применения композиционного правила вывода и методов дефазификации.

Опыт использования нечетких методов, приобретенный разработчиками фирмы Hitachi при выполнении этого проекта, привел к включению их разработки в продукт ES/KERNEL. Одна из самых больших японских компаний, занимающихся строительством и инженерно-строительными работами, использовала возможности нечетких множеств системы ES/KERNEL для построения системы поддержки принятия решений в процессе проектирования мостов. Главный японский банк в свое время также планировал использовать

эту технологию для создания сложной системы анализа финансовых схем с последующим вводом ее в повседневное использование. Применение нечетких множеств в экспертных системах-оболочках рассматривалось в [313, 314]. Система Object/IQ — это латинизированная версия системы ES/KERNEL, но уже без нечетких возможностей. Система XShell компании Expersoft из Сан-Диего — это набор объектно-ориентированных средств разработки, которые поддерживают нечеткие системы для управления процессами.

Реальные задачи, особенно задачи поддержки принятия решений, обычно требуют управления неопределенностью. По этой причине объектно-ориентированным методам необходимо понятие нечетких объектов и наследования в условиях неопределенности. Нечеткий объект может содержать одно нечеткое множество, определяющее значение какого-либо атрибута, или несколько нечетких атрибутов. В настоящее время проблема управления неопределенностью в объектно-ориентированных методах еще не решена. В системах наследования мы сталкиваемся с проблемой частичного наследования как четких, так и нечетких свойств. Эта задача оказывается особенно сложной при множественном наследовании. Здесь проблема состоит в том, как объединить унаследованные нечеткие свойства. Эта теория нуждается в отработке и согласовании. Приложение А — это маленький вклад в решение этой задачи.

Распределенные системы и системы клиент/сервер

Я рискнул предположить, что стремление к открытым распределенным системам приведет к потребности в объектной ориентации и к ее поддержке. Потребность возникнет в связи с тем, что структуры клиент/сервер, распределенные процессы и сложные пользовательские интерфейсы будут нуждаться в модельной выразительности и производительности, обеспечиваемой объектно-ориентированным подходом. Поддержку обеспечит пользовательский спрос на надежные, приспосабливаемые, дружественные системы, который появится после первых успехов объектно-ориентированных систем. Без повторного использования, расширяемости и семантического богатства нельзя создать по-настоящему открытые распределенные системы при достаточно низких затратах. Это подтверждают почти все тенденции, обсуждавшиеся в этой книге: язык программного интерфейса приложений, программное обеспечение среднего уровня, подходы CBD, EAI, Catalysis и т.д. Я также предсказывал, что принципы построения программного обеспечения среднего уровня и метод OMG сыграют важную роль в этой области, что и произошло на самом деле, как мы видели в главе 4.

Параллельные системы

Это область, где произошел относительно малый (но не нулевой) сдвиг со времени последнего издания этой книги, и исследовательская деятельность продолжается достаточно интенсивно.

Формальные методы

Надо отметить, что формальные методы связаны с проблемами неопределенности в спецификациях и проблемами соответствия семантики спецификаций семантике исходного кода. Хотя последнее вполне похвально, может существовать два подхода к неопределенности: она должна быть устранена при помощи более точной формализации или ее следует считать существенным компонентом самого описания. Например, каждый пользователь знает, что такое *достаточное* время ответа, но для полного устранения неопределенности формальный метод требует его выражения в точном количестве секунд (или в виде функции, возвращающей

число секунд). Более серьезная проблема возникает при формализации требования "дружественного пользовательского интерфейса". Я не могу представить себе разумного способа формализации дружественности и склоняюсь к мнению, что неопределенность — это существенная черта спецификации и ее саму следует включить в формализм. Эта точка зрения представлена в подходе к семантике наследования в приложении А.

Алан Тьюринг (Alan Turing) предложил использовать формальную логику для программирования компьютеров уже в 1948 году. В конце 1960-х эту идею подхватили Дейкстра (Dijkstra) и Хоаре (Hoare), которые показали, что семантику языка программирования и спецификации можно охарактеризовать формально. Системы формального доказательства дают возможность математикам работать с формальными спецификациями, основанными на логике или теории множеств, и фрагментами исходного кода и доказывать, что код соответствует спецификации. Более сложный подход, реализуемый такими методами, как Z или VDM, начинает работу с формальной спецификации, а затем пошагово преобразует ее в программу. Сложность этого подхода состоит в том, что он требует высокого мастерства в области математики и большого количества времени для разработки доказательств корректности. Уровень подготовки специалистов часто должен быть не ниже уровня кандидата физико-математических наук.

Метод VDM (Vienna Development Method) от компании IBM в настоящее время прошел процедуру стандартизации Британского института стандартов, и систему Z также планируется стандартизировать. К числу других методов и языков формальной спецификации относятся методы OBJ и *me too* от компании STC. Языки логического программирования также можно использовать для непосредственного получения выполняемых спецификаций. Но здесь существуют две проблемы. Во-первых, технология FOPC недостаточно выразительна для описания спецификации, например, если имеется неопределенность. Во-вторых, и это более серьезно, логическое программирование — это всего лишь программирование, и этот подход просто смещает этап кодирования на более ранние стадии жизненного цикла. Формальная логика сама по себе не свободна от проектирования, так как сам процесс постановки логической задачи включает в себя обращение к определенным положениям. Вот почему существует несколько разных логик. Еще одна проблема — это необходимость переобучать программистов формальной логике, овладеть которой — дело хитрое и непростое. Самый сложный подход основан на искусственном интеллекте и теории доказательств. Его философия состоит в том, что программа — это доказательство теоремы, представляющей собой спецификацию. Программы автоматического доказательства теорем находят подтверждение для спецификации. Когда доказательство завершено, полученный результат — это выполняемый код. Самые удачные методы доказательства теорем основаны на нестандартной логике, в том числе интуиционистской теории типов [525, 526]. В настоящее время (и так будет в ближайшем будущем) этот метод применим только для тривиальных небольших задач.

Поскольку формальные методы требуют больших затрат времени и высокого мастерства, их обычно считают достойными внимания только в областях, где решающую роль играет надежность. Однако идеи формальных методов можно применять неформально со значительной практической выгодой в коммерческих системах, особенно в комбинации с объектно-ориентированными методами.

Формальная спецификация отражается в объектно-ориентированном проектировании через понятие соглашения между объектом-сервером и его клиентом. Как было описано в предыдущих главах, это соглашение выражается в форме правил, инвариантов, пред- и постусловий. Чтобы использовать эту прекрасную идею, не требуется никакой формальной теории.

В [790] формальные методы применены для решения задач, связанных с использованием больших библиотек объектов. Основная идея состоит в том, что программисту надо знать, что делают объекты в библиотеке. Тогда почему бы не связать с этим объектом формальную спецификацию, на основе которой можно было доказывать теоремы и даже создавать составные объекты? Эта работа основывается на введении пред- и постусловий и эффективном расширении языка Smalltalk с учетом методов VDM и Larch [348]. При этом к языку Smalltalk добавляются свойства, которые считаются стандартными для языка Eiffel. Многие из этих идей в зрелой форме появились в методе Catalysis.

Смерть универсальной вычислительной машины

Слухи о гибели универсальной вычислительной машины, как по Марку Твену, были сильно преувеличены. Я отстаивал точку зрения, что роль универсальных вычислительных машин уменьшится во всем, кроме серверов данных. Это утверждение аргументировалось относительно невысокой производительностью машин, больших и малых. Однако оно давно устарело в связи с законом Мура (Moore)[1] и относительным снижением цен на универсальные вычислительные машины. Я также указывал на определенные приложения — такие как управление сетями ATM — в которых пропускная способность канала обеспечивается большими машинами и производительность не является ключевым требованием. Такова ситуация на сегодняшний день.

В заключение, сделав в прошлом так много неверных предсказаний, я осмелюсь сделать только одно. Я полагаю, что через несколько лет (или даже меньше) рынок завоюет новый язык. Может быть, синтаксически он будет похож на язык Java, но во всем остальном он будет сильно напоминать язык Eiffel.

10.5. Резюме

Электронная торговля — это суперприложение для объектной технологии. Она подразумевает технологию BPR. Ей необходимы методы CBD, согласованная реализация и компонентная архитектура.

Любое приложение можно разработать с помощью объектно-ориентированных методов. Мы вкратце рассмотрели графические пользовательские интерфейсы, моделирование, ГИС, параллельные системы, экспертные системы, модели "классной доски" и нейронные сети. Более подробно мы рассмотрели агентные системы, заметив, что объектно-ориентированные методы можно использовать для моделирования агентов, если в объекты можно ввести наборы правил, и что агенты являются удобным представлением для моделирования бизнес-процессов. Наконец, я пересмотрел некоторые свои прошлые прогнозы и сделал несколько новых.

[1] Закон Мура утверждает, что производительность компьютеров удваивается приблизительно каждые 1,5 года. Эта тенденция должна сохраниться, по крайней мере, до 2006 года.

10.6. Дополнительная литература

Существует слишком много книг по "всемирной паутине", графическим пользовательским интерфейсам, искусственному интеллекту и экспертным системам, которые следовало бы упомянуть здесь. Книга [629] является самым лучшим введением в географические информационные системы и содержит несколько конструктивных статей на эту тему. Работа [280] дает первое представление о принципах параллельного программирования в целом и об идеях в частности, на которых основан язык Strand. В [361] обсуждается архитектурная поддержка объектно-ориентированного программирования, предоставляемая машиной Rekursiv и ее языком системного программирования Lingo. В [331] представлены некоторые из моих собственных взглядов на экспертные системы, и это хороший библиографический источник, хотя сейчас он уже распродан. Работа [267] рассказывает о нескольких актуальных областях применения искусственного интеллекта и его преимуществах. В [752] рассматриваются некоторые системы программирования искусственного интеллекта. Работа [261] — это всесторонний литературный источник по системам "классной доски". Существенная ссылка по вопросу нейронных сетей — это [677]. Понятие фреймов искусственного интеллекта объясняется в [801] и очень ясно изложено в [697].

Первая полезная работа о распределенном искусственном интеллекте — это [398]. Отличное введение в тематику мультиагентных систем представляет [268]. В [576] обсуждается, как разрешать конфликты между различными агентами. Отличный и очень легко читаемый отчет о создании интеллектуальных агентов на языке Java — [77].

Некоторые из приложений, упоминаемых в этой главе, рассмотрены в [795]. В [441] немного подробнее обсуждается проблема объектно-ориентированных пользовательских интерфейсов и описываются многие интерфейсные приложения и продукты. Работа [631] подробно описывает шесть приложений и содержит полезные замечания об удобстве различных языков и проблемах управления проектами. В [748] обсуждается 18 применений объектно-ориентированного подхода. В [442] описан объектно-ориентированный подход к автоматизации офисной деятельности с подробной информацией о мультимедийных средствах, системах клиент/сервер, сетях, графических пользовательских интерфейсах и программном обеспечении коллективного пользования. Работа [763] дает широкое представление о теории типов, которая кратко описана в [331], где также рассматриваются некоторые вопросы искусственного интеллекта и управления неопределенностью. В [497] кратко рассказывается о нескольких значительных коммерческих приложениях.

A

Нечеткие объекты: наследование в условиях неопределенности

Это приложение включено в рассмотрение для того, чтобы детальнее обсудить некоторые замечания, сделанные в главах 6 и 7, поскольку представленный здесь материал не так легко доступен в литературе. Для того чтобы понять оставшуюся часть книги, не обязательно изучать это приложение, и читатели, незнакомые с экспертными системами, теорией нечетких множеств или математикой, могут посчитать его труднодоступным.

В главе 6 было показано, что существует множество стратегий разрешения конфликтов при наследовании значений, непригодных для поддержки множественного наследования. Это техническое приложение описывает одну из наиболее необычных технологий, которая применяется для управления неопределенными данными, когда линии наследования сами по себе не могут быть реализованы с полной определенностью. Эта проблема обычно не возникает в процессе наследования методов, а если возникает, то требует очень сложного управления распространением факторов уверенности, связанных с возвращаемыми методами переменными. Таким образом, ограничимся изучением наследования признаков. Частичное наследование методов, представленное фактором уверенности, означает возможность выполнения операции объектом. Ясно, что это вряд ли имеет значение для программиста, но может быть очень полезно при описании систем в таких контекстах, как моделирование предприятий и обратное проектирование бизнес-процессов.

В этом приложении описан вычислительный метод представления неопределенного знания, который был разработан как обобщение понятия фрейма, введенного Минским [562], и концепции объекта из объектно-ориентированного программирования. Поскольку это обобщение основано на теории нечетких множеств Заде [823], естественно определять обобщенные объекты как *нечеткие* (fuzzy object). Эти структуры впервые были введены в контексте искусственного интеллекта (AI — Artificial Intelligence) в работе [330] под названием нечетких фреймов (frame). Позднее появились некоторые другие, более ограниченные понятия нечетких объектов, подобные введенному в [812].

Для начала вспомним версию объектов, используемую в системах AI и теории фреймов, а также ознакомимся с аппаратом теории нечетких множеств. В конце приложения исследуем некоторые интересные вопросы, которые возникают в теории, и предложим темы для дальнейших исследований. В процессе рассмотрения имеет смысл сравнить исследуемый подход с нечеткими кванторами Заде [826] и немонотонной логикой [536, 658]. Несложно предположить, что существуют важные связи исследуемых вопросов с моделированием семантических данных и объектно-ориентированными базами данных. Нечеткие объекты дают унифицированную основу для представления как определенного, так и неопределенного знания об объектах и в некотором смысле обобщают нечеткие отношения.

A.1. Представление знаний об объектах в искусственном интеллекте

Существует много способов представления знаний: в виде правил, с помощью логики, в виде процедур и т.д. Обычно каждый из этих формализмов лучше всего подходит для некоторого конкретного типа знания. Например, правила хороши для описания знаний о причинности, логические схемы — для описания отношений, а для описания знания о вычислении кубического корня не существует лучшего способа, чем процедура. С другой стороны, описание чаши с фруктами или окна с витражами в форме правил будет безнадежно скучным, если не сказать больше. Более полное описание способов представления знаний можно найти в [110, 697] или [330]. В AI для представления знаний об объектах и их свойствах (таких, как чаши с фруктами) обычно используются семантические сети и фреймы. Сосредоточим наше внимание на этих объектных представлениях.

Часто знание является неопределенным, и в экспертные системы для моделирования неопределенности обычно вводятся некоторые дополнительные механизмы. В подходах на основе правил это можно сделать путем присвоения правилам или их атомарным компонентам факторов уверенности или вероятностей или с помощью процедуры поддержки истинности, которая зависит от типа неопределенности. В [700] отмечено, что "программный агент не может обладать полным знанием о своем окружении (если оно не является тривиальным), и, следовательно, он должен уметь делать выводы на основании неполной и неопределенной информации".

Во фреймовых или объектно-ориентированных системах неопределенность часто возникает как побочный результат множественного наследования. В этом приложении мы рассмотрим типы неопределенности, которые легко промоделировать с помощью нечетких множеств, но в принципе предлагаемый подход применим и к стохастическим задачам.

А.1.1. Семантические сети и фреймы

Сначала необходимо изложить мою философскую позицию относительно понятия *фрейм* (frame). Я считаю фреймы структурами данных, а не моделями знания человека. Более того, на мой взгляд, между фреймами и структурно эквивалентными понятиями, такими как семантические сети, различие небольшое (если есть вообще). Это мнение поддерживают многие исследователи, работающие в области AI, например Брахман (Brachman), которого интересовали вопросы логической адекватности теории, а не практические вопросы, мотивирующие эту работу. Я не поддерживаю тех, кто, подобно Хайесу [366], сводит фреймы к логике первого порядка. На мой взгляд, основное назначение фрейма состоит в представлении конструкций более высокого порядка, в частности информации из объектно-ориентированных баз данных или семантических моделей данных.

Семантическая сеть состоит из множества узлов и набора упорядоченных пар узлов, называемых *связями*. Она также обеспечивает интерпретацию их значений. Я ограничусь представлением этой интерпретации на основе описательной семантики, т.е. множества утверждений, описывающих интерпретацию. Конечные узлы связей называются **слотами** (slot), если они представляют свойства (предикаты), а не объекты или классы объектов. Фреймом называется семантическая сеть, представляющая объект (или стереотип этого объекта). Она состоит из множества слотов и набора исходящих связей. Для прояснения ситуации рассмотрим фрейм игрушечного блока, изображенный на рис. А.1 в виде сети.

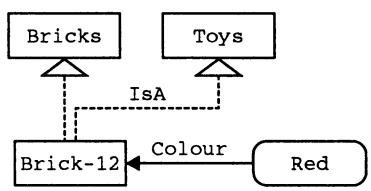


Рис. А.1. Фрейм, представленный в виде семантической сети

Его можно представить в табличной форме следующим образом.

```
Brick-12
IsA:        Brick,
            Toy
Colour:     Red
```

Набор фреймов, образующих семантическую сеть, будет рассматриваться в этом приложении как **основанный на объектах** (object-based). В приведенном выше примере неявно присутствуют фреймы для понятий `Brick` и `Toy`.

На рис. А.2 проиллюстрировано наследование свойств слотами. Это также можно представить в виде двух фреймов следующим образом.

```
Brick                                   Toy
AKO:        Block,Commodity             AKO:Commodity
Shape:      Cuboid
```

Фреймы могут наследовать свойства через связи IsA или AKO, так что Brick-12 наследует от объекта Brick значение слота Shape, а также те свойства игрушек и блоков, которые не вызывают противоречий. Связь IsA используется для обозначения принадлежности к классу, а AKO — для обозначения включения. В [761] отмечено, что попытки разграничения связей IsA и AKO предпринимаются при использовании множественного наследования. В нашем изложении мы не станем без необходимости акцентировать внимание на различии связей IsA и AKO [109].

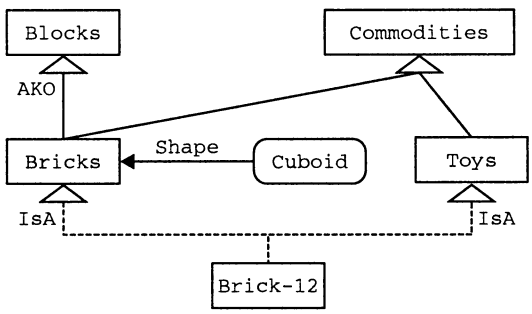


Рис. А.2. Множественное наследование свойств

В работе [801] вводятся производные слоты (**фацеты**), допускающие значения по умолчанию, демоны и перспективы. Эти элементы не рассматриваются здесь в применении к нечетким объектам, но для этого нет никаких принципиальных препятствий.

А.1.2. Наследование свойств

Наследование обеспечивает в компьютерных системах метод рассуждения с помощью неявных фактов. Как описано в главе 3, существуют различные языки программирования, основанные на фреймах, в том числе FRL, KRL, KEE, Leonardo и др. Однако большинство из них испытывает различные проблемы. Они обычно не имеют формальной семантики, не позволяют рассуждать об исключениях (немонотонная логика) и не реализуют частичное наследование ни в смысле частичного наследования свойства, ни в смысле наследования комбинации частично верных свойств. Другие авторы обсуждали частичное наследование, но с совершенно иной точки зрения. Примером является работа [440]. Задача этого приложения состоит в том, чтобы решить все эти проблемы в рамках единого унифицированного подхода.

В работе [761] отмечено, что немонотонная логика, или логика умолчания, обладая формальной семантикой, является слишком общей для практических целей, так как не имеет средств реализации наследования, позволяющих выполнять рассуждения на неявных данных.

Следует отметить двойственность между логиками, в которых добавлены дополнительные операторы, такие как L и N в модальных логиках и M в немонотонных логиках, и логиками, расширяющими пространство истинности, к которым относятся многозначная, нечеткая и вероятностная логики. В этом приложении рассматривается нечеткая логика Заде с сигма-счетной интерпретацией семантики [432, 826].

В [700] доказано, что системы фреймов должны поддерживать так называемые рассуждения "по очевидности". Решение, предложенное Шастри (Shastri), не требует немонотонности

или нечетких значений истинности. Его подход включает семантику подсчета частоты. Однако я считаю, что нечеткие значения истинности приносят определенную пользу, и допускаю право на существование обоих подходов.

Позднее будет неформально показано, как механизм нечетких объектов можно использовать для преодоления некоторых проблем немонотонных рассуждений, которые часто возникают в системах наследования или при реализации полиморфизма.

Нечеткие объекты обеспечивают вычислительно эффективные средства моделирования систем поддержки истинности или возможных миров без введения модальных операторов.

Для дальнейшего изложения потребуется аппарат нечеткой логики, который вводится максимально кратко. Для читателя, не обладающего базовыми знаниями из области нечетких множеств, предлагаем ознакомиться с работой [330], где приводится объяснение всех используемых здесь понятий.

А.2. Основные понятия теории нечетких множеств

Понятие нечеткого множества введено Заде в 1965 году. Оно включает ослабление ограничения на характеристическую функцию множества — теперь она может принимать не только два значения. В этом разделе кратко рассматриваются принципы теории нечетких множеств, которая понадобится позже в этом приложении. Это делается просто для согласования терминологии, а не с учебной целью. Более полную информацию можно найти в [330] или [432].

А.2.1. Нечеткие множества

Нечетким множеством является функция $f:X \rightarrow I$, отображающая элементы множества X в значения из единичного интервала I. Эти значения можно интерпретировать как лингвистические значения для переменных из области определения. Например, если область определения X представляет уровень благосостояния (на произвольной денежной шкале), то можно определить нечеткие множества, определяющие нечеткие лингвистические термы "богатый", "зажиточный" и "бедный" (рис. А.3). Единичный интервал (по вертикальной оси) используется для того, чтобы представить степень истинности, так что понятие "бедный" полностью соответствует нулевым накоплениям. Степень принадлежности к множеству бедных людей уменьшается с ростом благосостояния до тех пор, пока не будет достигнута точка нулевой степени принадлежности (или нулевой истинности). Конечно, можно рассматривать другие термы, такие как "жутко богатый" или "в конец обедневший", но это не заметки о социальной политике. Нечеткие множества удобно представлять графически. Они также могут быть представлены векторами значений истинности.

Существует несколько видов нечеткой логики. В одной из стандартных логик, представленных здесь, операции исчисления высказываний определены для нечетких предикатов следующим образом.

$$f \text{ AND } g = min(f,g),$$
$$f \text{ OR } g = max(f,g),$$
$$\text{NOT } f = 1\text{-}f.$$

Импликация определяется обычным способом, а именно:

$$(f \Rightarrow g) = (\text{NOT } f) \text{ OR } g = max(1\text{-}f, g).$$

Множество термов (допустимых лингвистических значений) можно расширить, используя пропозициональные операторы и операции, известные под названием ограничителей. В качестве примеров рассмотрим ограничители "очень" и "достаточно", часто определяемые как:

$$VERY\, f = f^2, \qquad QUITE\, f = \sqrt{f}.$$

Таким образом, выражение "очень богатый или не очень бедный" можно интерпретировать как нечеткое множество. Это очень удобно для представления знаний и используется во многих других работах.

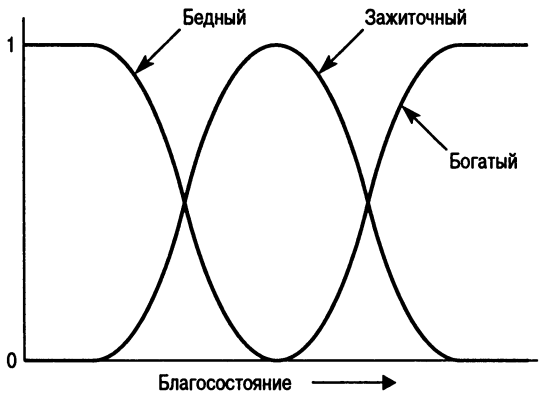


Рис. А.3. Множество нечетких термов для переменной "благосостояние"

А.2.2. Правила логического вывода

Учитывая эти основные определения, рассмотрим проблему представления неточного вывода с помощью нечетких множеств. При этом будем использовать утверждения вида *X есть A* и правила

$$Если\ X\ есть\ [не]\ A\ [и|или\ X'\ есть\ ...],\ то\ Y\ есть\ B,$$

где *X*, *X'* и *Y* обозначают объекты, а *A* и *B* — нечеткие множества.

Простые силлогизмы, такие как правило *модус поненс*

$$\frac{\begin{array}{c} X\ есть\ A \\ Если\ X\ есть\ A,\ то\ Y\ есть\ B \end{array}}{Y\ есть\ B},$$

обрабатываются следующим образом. Вычисляется декартово произведение всех лингвистических переменных, выступающих в роли предположений, и берется их пересечение *E*.

Декартово произведение и пересечение двух нечетких множеств показаны на рис. А.4. Это нечеткое множество интерпретируется как гибкое ограничение на решение. Затем для каждого следующего дизъюнкта в каждом правиле текущее скалярное значение переменной *X* используется для определения уровня истинности предшествующего нечеткого множества, так что нечеткое множество *B* эффективно усекается. Результирующее усеченное нечеткое множество формируется как объединение с множеством ограничений *E*. В этом приложении будет использоваться именно это правило, хотя были предложены и другие.

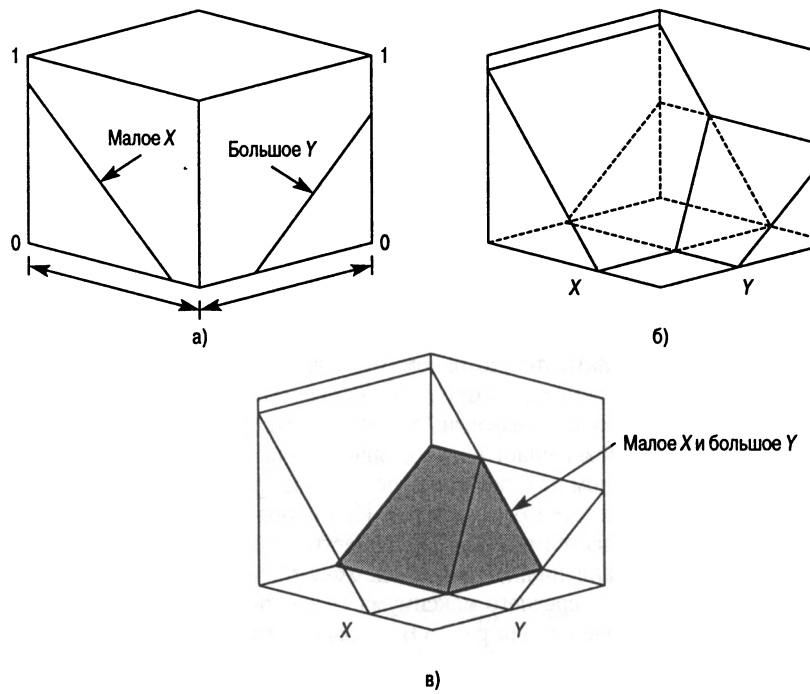


Рис. А.4. Нечеткие множества для двух различных областей определения (а), декартово произведение двух нечетких множеств (б) и пересечение двух нечетких множеств (в)

В качестве простого примера нечеткого вывода рассмотрим правило

Если X есть A, то Y есть B.

Нечеткие множества *A* и *B* изображены на рис. А.5. Если входным значением является *x* из области определения *X*, то первый шаг процедуры логического вывода состоит в определении значения истинности *x* или его степени принадлежности множеству *A*. Из рис. А.5, *а* видно, что оно равно 0,6. Следовательно, нечеткое множество *B* усекается на этом уровне. Результатом логического вывода является усеченное нечеткое множество, которое на рис. А.5, *б* заштриховано.

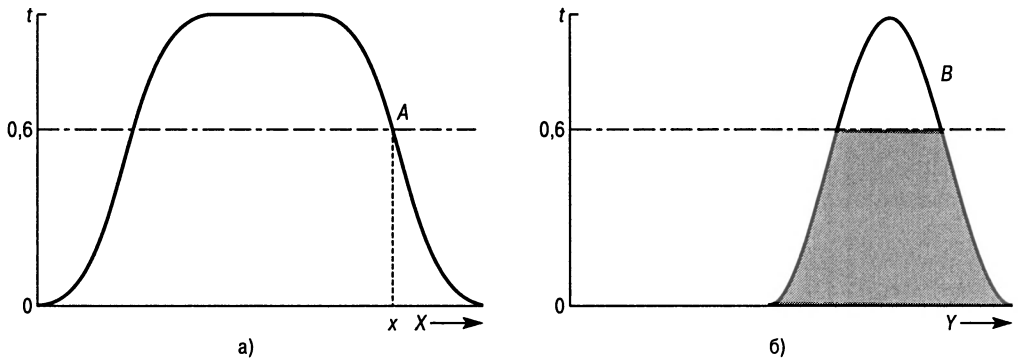


Рис. А.5. Степень принадлежности (или значение истинности) **x** нечеткому множеству **A** составляет 0,6 (а). Нечеткое множество **B** усекается на уровне истинности **x** (б)

А.2.3. Дефаззификация

Если в качестве выходных значений неудобно использовать нечеткие множества, прибегают к процедуре дефаззификации, позволяющей вернуться к скалярному значению. Существует множество способов сделать это. Познакомимся с двумя из них. Метод "среднего максимума" (или максимума) подразумевает возвращение арифметического максимума скалярного значения из области определения результирующего нечеткого множества. Метод "центра моментов" (или моментов) возвращает среднее значение из области определения с учетом значения истинности в выходном нечетком множестве — центр тяжести воображаемого картонного очертания графика нечеткого множества. Использование этих методов в различных приложениях широко обсуждается в [331]. По существу, правило моментов является наилучшим для приложений управления, где на выходе желательно получить гладкое изменение выходного значения, а правило среднего максимума — для приложений, где предпочтительны дискретные выходные значения. На рис. А.6 показано, что для любого несимметричного нечеткого множества эти два правила возвращают различные скалярные величины.

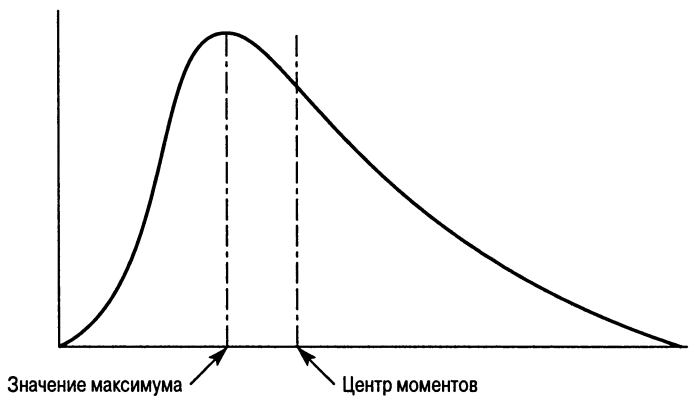


Рис. А.6. Два метода дефаззификации

Если в качестве выхода требуется нечеткое множество, но желательна некоторая регулярность его формы, то можно обратиться к методу, известному как лингвистическая аппроксимация. Этот процесс включает предопределение допустимого *множества термов* нечетких множеств на области определения (рис. А.3). В случае нечетких чисел (нечетких подмножеств вещественной прямой) примером множества термов может служить множество {*крошечный, маленький, средний, большой, огромный*}. Множество термов может быть расширено с помощью операций над нечеткими множествами (т.е. за счет понятий "не очень маленький"). Лингвистическая аппроксимация возвращает терм, "наиболее близкий" результирующему нечеткому множеству в соответствии с некоторой утвержденной мерой расстояния.

А.2.4. Нечеткие кванторы

В отличие от четких кванторов "для любого" и "существует", нечеткие кванторы представляются такими словами, как "большинство", "почти все", "некоторые" и т.д. Они часто используются в обыденном языке. Таким образом, выражение "птицы летают" может интерпретироваться как "большинство птиц могут летать". Заде вводит правила логического вывода и формальной семантики для таких утверждений. В его семантике кванторы интерпретируются как гибкие ограничения на множество нечетких отношений, которые считаются сущностями. Эта структура соответствует отношениям объектов в мире (базе данных). Типичным правилом вывода для этой системы является

$$\frac{\begin{array}{l} Q1\ A\ есть\ B \\ Q2\ (A\ и\ B)\ есть\ C \end{array}}{Q1 * Q2\ A\ есть\ (B\ и\ C)},$$

где Q — это нечеткие кванторы, интерпретируемые как нечеткие числа, а $*$ обозначает их произведение. Например, этому правилу удовлетворяет силлогизм.

$$\frac{\begin{array}{l} Большинство\ (около\ 90\%)\ птиц\ могут\ летать \\ Большинство\ (около\ 90\%)\ летающих\ птиц\ имеют\ перья \end{array}}{Достаточно\ много\ (около\ 81\%)\ птиц\ имеют\ перья}$$

Конечно, существуют другие правила логического вывода. Точное значение теста зависит от меры, используемой для измерения мощности нечеткого множества. Обычно используется сигма-счетная мера, которая является арифметической суммой степеней принадлежности (в непрерывном случае — интегралом). Более детальное описание может быть найдено в [432].

На этом завершается рассмотрение понятий из теории нечетких множеств, которые потребуются в этом приложении.

A.3. Нечеткие объекты

Перейдем к основному материалу этого приложения. Понятие объекта расширяется на понятие нечеткого объекта двумя способами: за счет использования нечетких множеств в качестве значений атрибутов, помимо текстовых, списочных и числовых значений[1], либо за счет частичного наследования через атрибуты AKO или `IsA`. Далее будет показано, как обобщить эти подходы в третьем направлении, допуская, чтобы экземпляры содержали по нескольку наборов значений атрибутов. Это обеспечит представление возможных миров или изменяющихся во времени объектов.

Чтобы объяснить преимущества и механизм работы нечетких объектов, не будем развивать формальные математические рассуждения, а воспользуемся конкретным примером. Рассмотрим следующую задачу. Вы, как покупатель, хотите оценить безопасность товаров для отдыха. Объекты обеспечивают способы представления знаний и данных, касающихся ключевых абстракций или понятий из предметной области. Наиболее естественным путем анализа проблемы (памятуя, что мы собираемся создать компьютеризированный советчик) является перечисление задействованных физических объектов. Предположим, что среди множества других товаров нужно оценить безопасность лодки и планера. Эти объекты являются представителями более общих объектов и связаны с ними через различные атрибуты, методы и правила. Методы не относятся к данному обсуждению. На рис. A.7 изображен способ представления знаний о некоторых объектах, представляющий интерес в контексте использования нечетких объектов. Отметим, что экземпляры отличаются от классов использованием атрибута `IsA` вместо AKO. Кроме того, они обозначаются прямоугольниками с острыми углами. Обозначение нечетких классов в виде прямоугольника с округленными углами в точности соответствует стереотипу UML `<<fuzzy>>` и используется для удобства представления. Атрибут AKO определяет суперкласс. Теперь в процессе решения задачи рассмотрим девять из этих объектов, в каждом случае снабжая их объяснением синтаксических соглашений. Наиболее общим объектом является предмет потребления `Commodity`.

На этом рисунке можно выделить основные и случайные классы и экземпляры. Первые шесть являются основными. Кстати, отметим синтаксическое обозначение для представления значений по умолчанию и демонов обратных цепочек (процедуры `IfNeeded`), а также демонов прямых цепочек (`IfUpdated`). Способ применения правил для интерпретации обозначений станет понятен позднее.

Рассмотрим понятие средства передвижения `Vehicles`. Общие знания о его атрибутах представлены на рис. A.8.

Здесь атрибут `Superclasses` указывает на другой объект, в данном случае на наиболее общий из возможных. При этом используется стереотип UML. Наряду с другими изображенными структурами, средство передвижения является нечетким объектом по двум причинам. Во-первых, степень наследования свойства от объекта, определяемого атрибутом `Superclasses`, можно задать числом от 0 до 1 в квадратных скобках после имени. На рисунке не указано никакого значения, и предполагается значение по умолчанию [1,00]. Во-вторых, другие атрибуты в качестве значений могут содержать нечеткие множества (векторы значений истинности). Выражения в скобках определяют тип значения: `[fuzz]`, `[real]`, `[list]`, `[text]` или некоторый определенный пользователем абстрактный тип данных.

[1] Это не является настоящим расширением, поскольку нечеткие множества составляют абстрактный тип данных. Это можно считать расширением только в том случае, если тип данных считать примитивом.

Нечеткое множество `high` (высокий), используемое в объекте `Vehicles`, может быть представлено функцией принадлежности, показанной на рис. А.9. Поскольку это класс, данное значение (`high`) задается по умолчанию.

Для общего класса предметов потребления можно задать определенные значения атрибутов, добавив соответствующий объект. Предположим, для предмета потребления `Commodity` атрибуты `Safety` (безопасность) и `Utility` (полезность) принимают значение `high`.

Естественно предположить, что, если покупка является безопасной, объект `Toy` (игрушка) будет наследовать значение `high` атрибута `Safety`, поскольку значение функции принадлежности нечеткому множеству для этого атрибута на всем интервале равно 1. Значения других атрибутов не изменяются. Наследуются лишь значения атрибутов ближайшего предка, и то лишь в том случае, если значение соответствующего атрибута дочернего объекта не определено. В некоторых приложениях может понадобиться наследование даже для тех атрибутов, которые содержат определенные значения. В этом случае механизм наследования модифицируется — берется пересечение (минимальное значение) нечетких множеств родительского и производного объектов. Этот принцип можно назвать *нечетким предположением о замкнутости мира* (FCWA — Fuzzy Closed World Assumption). Таким образом, если значения атрибутов представляют неизмененное знание о состоянии мира и его ограничениях, то не стоит допускать противоречивых операций присваивания новых значений, не учитывающих значения соответствующих атрибутов родительского объекта. Это интересный вариант перекрытия, основанный на совместном использовании информации и компромиссе между родительским и дочерним объектами. Хранилищем этой управляющей информации является раздел *правил* объекта, где инкапсулирована декларативная семантика.

Обсудим два других объекта, представляющих общие классы. Это вспомогательные объекты `Borrowed-object` (Заимствованный объект) и `Dangerous-object` (Опасный объект).

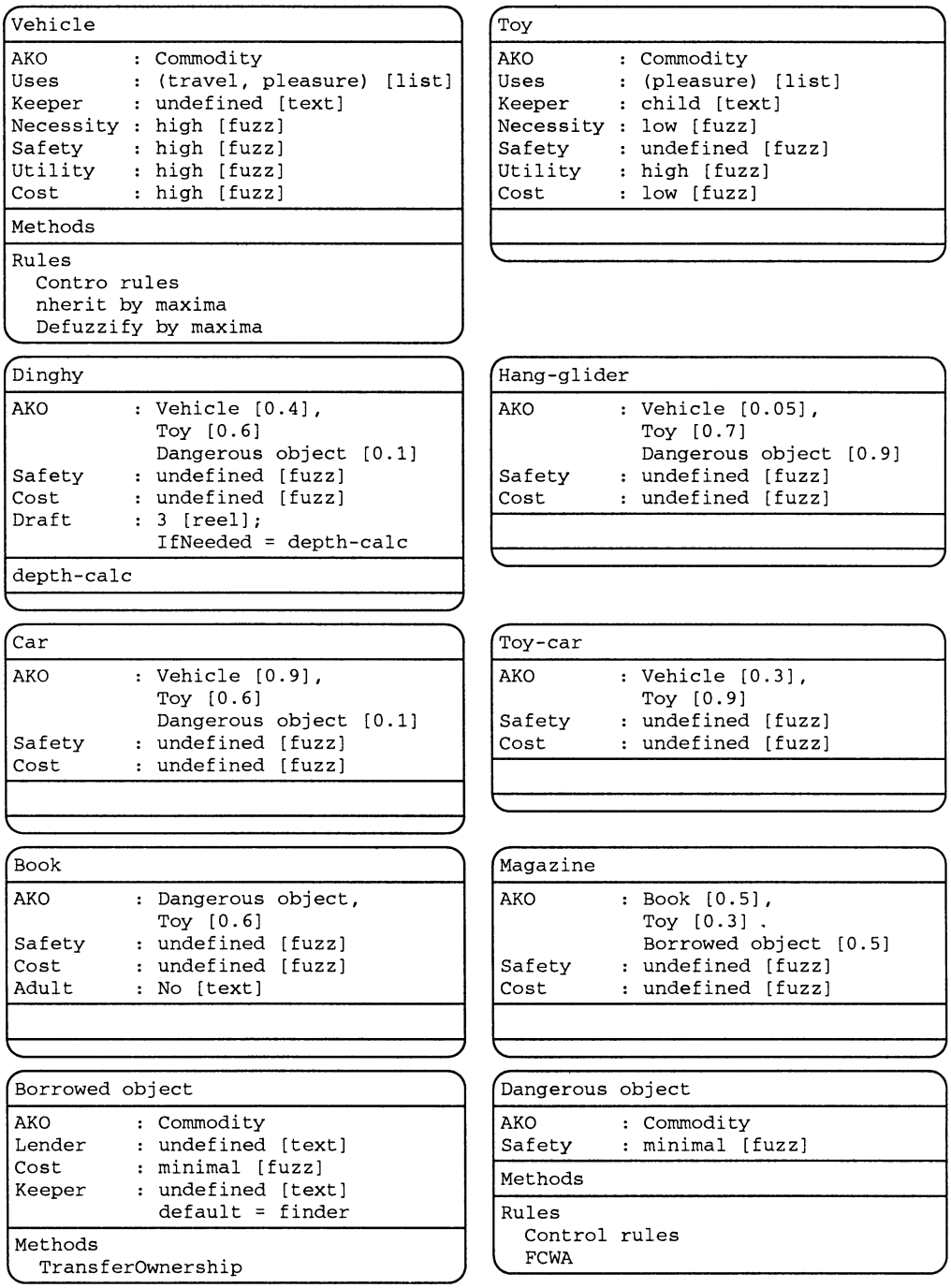
Здесь можно рассмотреть грустную возможность неявного заимствования объекта, такого как книга.[2] Поэтому, в случае заимствования объекта `Magazine` (Журнал), свойство собственности объекта `Borrowed-object` можно наследовать только наполовину. Механизм наследования добавляет фактор уверенности 0,5 к значениям `Lender` и `Keeper` (если они известны). Механизм наследования нечетких атрибутов состоит в том, что нечеткие множества (в этом случае `minimal`) усекаются на уровне 0,5. Возвращаясь к основной канве нашего изложения, заметим, что были введены два новых нечетких множества, определение которых графически представлено на рис. А.10.

Теперь мы подошли к объектам, описывающим достаточно специфические элементы схемы. Например, рассмотрим объекты `Dinghy` (лодка) и `Hang-glider` (планер). Конечно, для них можно определить абсолютно четкие атрибуты, такие как размах крыла планера.

Однако необходимо понять, какие значения могут принимать неопределенные атрибуты объектов самого низкого уровня (экземпляров) `Dinghy-123` и `Hang-glider-765` (рис. А.11). Сначала следует обратить внимание на "четкий" атрибут `Draft` и множественное наследование объектов высших уровней. Рассмотрим атрибут `Safety` объекта `Dinghy`.

---

[2] Другими словами, быть (вежливо) украденным!


Vehicle
AKO : Commodity
Uses : (travel, pleasure) [list]
Keeper : undefined [text]
Necessity : high [fuzz]
Safety : high [fuzz]
Utility : high [fuzz]
Cost : high [fuzz]
Methods
Rules
Contro rules
nherit by maxima
Defuzzify by maxima
Toy
AKO : Commodity
Uses : (pleasure) [list]
Keeper : child [text]
Necessity : low [fuzz]
Safety : undefined [fuzz]
Utility : high [fuzz]
Cost : low [fuzz]
Dinghy
AKO : Vehicle [0.4],
Toy [0.6]
Dangerous object [0.1]
Safety : undefined [fuzz]
Cost : undefined [fuzz]
Draft : 3 [reel];
IfNeeded = depth-calc
depth-calc
Hang-glider
AKO : Vehicle [0.05],
Toy [0.7]
Dangerous object [0.9]
Safety : undefined [fuzz]
Cost : undefined [fuzz]
Car
AKO : Vehicle [0.9],
Toy [0.6]
Dangerous object [0.1]
Safety : undefined [fuzz]
Cost : undefined [fuzz]
Toy-car
AKO : Vehicle [0.3],
Toy [0.9]
Safety : undefined [fuzz]
Cost : undefined [fuzz]
Book
AKO : Dangerous object,
Toy [0.6]
Safety : undefined [fuzz]
Cost : undefined [fuzz]
Adult : No [text]
Magazine
AKO : Book [0.5],
Toy [0.3] .
Borrowed object [0.5]
Safety : undefined [fuzz]
Cost : undefined [fuzz]
Borrowed object
AKO : Commodity
Lender : undefined [text]
Cost : minimal [fuzz]
Keeper : undefined [text]
default = finder
Methods
TransferOwnership
Dangerous object
AKO : Commodity
Safety : minimal [fuzz]
Methods
Rules
Control rules
FCWA

Dinghy-123	
IsA	:Dinghy
Draft	: undefined [real]
Safety	: undefined [fuzz]
Cost	: undefined [fuzz]

Hang-glider-765	
IsA	: Hang-glider, Borrowed object
Safety	: undefined [fuzz]
Cost	: undefined [fuzz]

Рис. А.7. Некоторые нечеткие объекты

Vehicles <<fuzzy>>
Superclasses: Commodities
Uses (text, 0, n) : (travel, pleasure) Keeper (text, 0, 1) Necessity (fuzz, 1, 1) : high Safety (fuzz, 1, 1) : high Utility (fuzz, 1, 1) : high Cost (fuzz, 1, 1) : high

Рис. А.8. Нечеткий класс или тип `Vehicles`

Рис. А.9. Нечеткое множество, представляющее понятие "высокий" на произвольной интервальной шкале

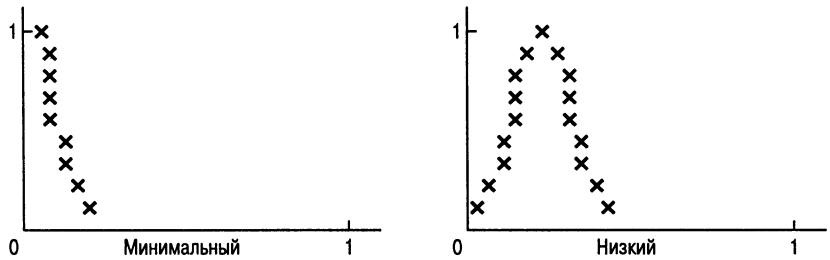


Рис. А.10. Нечеткие множества `minimal` и `low`

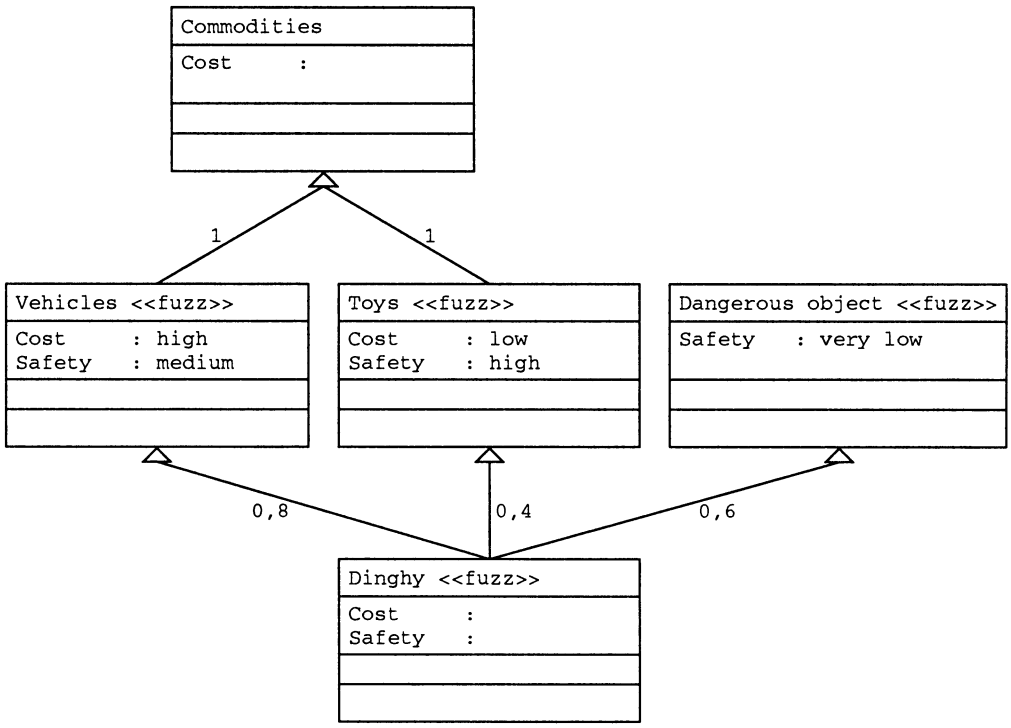


Рис. А.11. Сеть частичного наследования для объекта `Hang-glider-765`

Поскольку лодка `Dinghy` является средством передвижения, ее атрибут `Safety` наследует значение `high`. Но это верно только до уровня 0,4, поэтому нечеткое множество усекается на этом уровне. Этот объект также наследует значение `minimal` от объекта `Dangerous-object`, но только до уровня 0,1. Путь наследования от объекта `Commodity` через `Toy` дает значение `high` со степенью уверенности 0,6. Эти нечеткие множества объединяются с помощью оператора объединения, как показано на диаграмме на рис. А.12, *б*. Если бы это был окончательный результат некоторого процесса рассуждений, результирующее нечеткое множество было бы дефаззифицировано (в данном случае с помощью операции усреднения максимума) и была бы получена степень истинности для терма `safe`. В качестве альтернативы можно использовать лингвистическую аппроксимацию, позволяющую вернуть слово, соответствующее нормальному, выпуклому нечеткому множеству[3], представляющему возвращаемое значение. В других приложениях можно применять метод дефаззификации на основе моментов. На рис. А.12, *б* изображено нечеткое множество для представления стоимости лодки. Конкретная модель `Dinghy-123` наследует оба эти значения.

[3] Другими словами, нечеткое множество, которое достигает значения 1 (нормальное) и имеет только один пик (выпуклое). Нечеткие множества, показанные на рис. А.3, А.5, а, А.6, А.8 и А.9, обладают обоими свойствами.

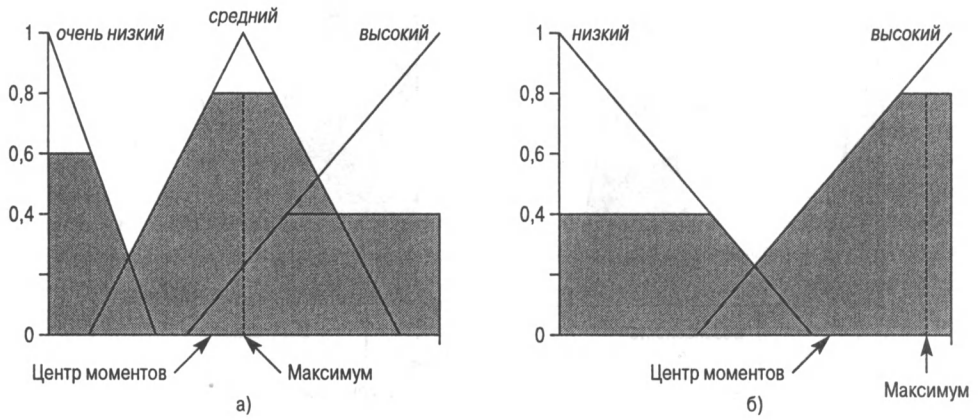


Рис. А.12. Математический аппарат частичного наследования для классов `Safety` (а) и `Cost` (б)

Стоимость здесь интерпретируется как цена, которую хотели бы уплатить. Поэтому стоимость покупаемой лодки обычно является низкой. Гораздо интереснее рассмотреть атрибут безопасности объекта `Hang-glider` (рис. А.13, *а*).

Здесь атрибут `Safety` наследует значение `minimal` от нечеткого множества для объекта `Dangerous-object` [0,9] и значение `high` — от обоих объектов `Vehicle` [0,05] и `Commodity` (через `Toy`) [0,7]. Применяя операцию объединения или дизъюнкции к этим трем нечетким множествам, чтобы показать, что атрибуты AKO являются *альтернативными* точками зрения на объект, получим результирующее нечеткое множество. В результате дефазификации получим значение, близкое к нулю (или лингвистической аппроксимации терма `minimal`).

Таким образом, система способна сделать корректный вывод, что планер — это очень опасная модель, которая при этом не слишком дорога.

Выбор оператора получения максимума может оказаться проблематичным в силу сложности присвоения определенной семантики операции дизъюнкции OR. В рассмотренном подходе этот выбор обеспечивается введением управляющего параметра или правил, которые позволят пользователям определить альтернативную операцию минимума. Это позволит реализовать приложения, в которых естественная интерпретация комбинации значений является разной и, возможно, представляет вероятностную точку зрения о том, что наследовать нужно только те значения, которые подтверждаются всеми источниками. При этом может понадобиться расширить множество типов правил в пределах блока правил объекта. Об этом речь пойдет в разделе А.6.

Данные рассуждения основаны на точке зрения, что планер — это безопасное средство передвижения из пункта А в пункт В и, конечно, обратно. Этот факт связан с предположениями об опасности объектов. Он может вызвать проблемы при проектировании нечетких объектных баз данных. Однако существует еще одна проблема. Если (что вполне обосновано) большинство людей считают планер игрушкой (с уверенностью 0,95), то результаты применения правила максимума в рамках логического вывода будут отличаться от интуитивно полученных результатов. Планеры следует признать высоко безопасными. Очевидная интуитивная неадекватность такого вывода объясняется неполнотой этого примера.

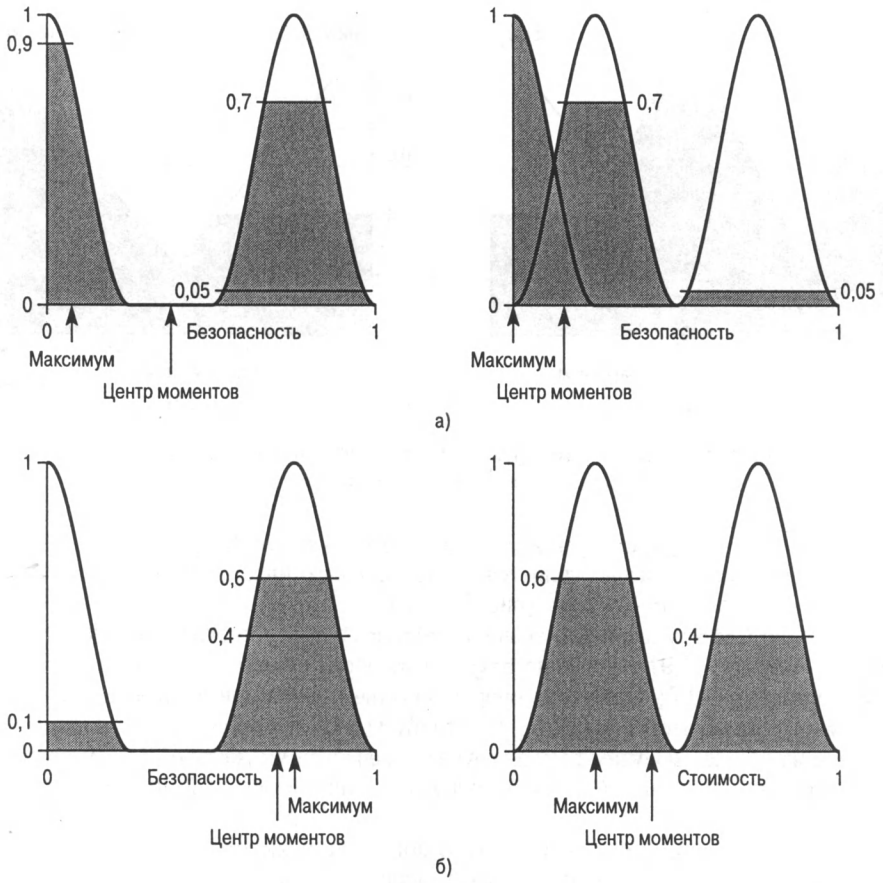


Рис. А.13. Унаследованные нечеткие множества для объектов `Hang-glider-765` (а) и `Dinghy-123` (б)

Стоит отметить, что подобная дилемма связана со всеми объектно-ориентированными системами фреймов и множественного наследования. Некоторое свойство, такое как опасность, можно связать с конкретным объектом либо через наследование, либо через его включение в дескриптор объекта. Путь обхода этой проблемы предложен в разделе А.7. В настоящее время определение критерия для нечетких и даже четких семантических моделей представляет собой тему отдельного исследования. Ожидаемый результат является аналогом теории нормальных форм в разработке баз данных.

Временно отложим этот вопрос и исследуем более интуитивно понятный пример. Описания типов опустим, так как они всегда ясны из контекста.

Механизм наследования, показанный на рис. А.14, позволяет сделать вывод, что объект `JohnPaulGettyIII` описывает симпатичного парня, о котором будут помнить, поскольку филантропы обычно известны. С другой стороны, `Apollo` наследует средние умственные способности по линии людей и всеведение по линии богов. От Гомера известно, что ум Аполлона

проявлялся только в некоторых ситуациях, что отражено в возвращаемом нечетком множестве его интеллекта, лингвистическая аппроксимация которого соответствует термину "яркий". Великодушие Аполлона также унаследовано по линии людей. Из-за отсутствия более богатой структуры или, другими словами, большего знания и информации можно сделать вывод только о среднем уме JohnPaulGettyIII, хотя последующие поколения могут помнить о нем как о яркой личности, благодаря его бессмертию (в смысле жизни в памяти). Ясно, что этот тип объектов имеет важное применение в компьютеризованных моделях рассуждений на основе здравого смысла. Заде [827] называет этот тип обоснования обычным, в то время как Турецкий [761] (ошибочно) именует его "нормативным" обоснованием (имея в виду нормальное).

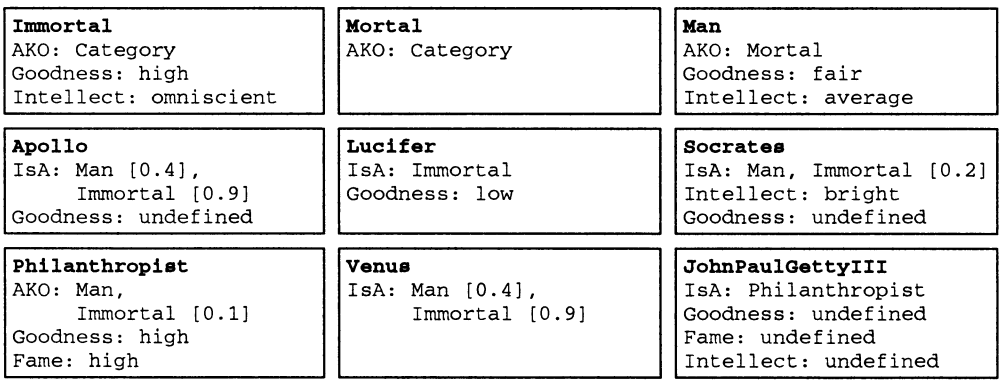


Рис. А.14. Некоторые другие нечеткие объекты

В этом примере затронуто много вопросов наследования, актуальных и для четких систем. Однако здесь речь идет о более сложной объектной базе. В частности, в конкретных приложениях можно потребовать, чтобы при наследовании божественных свойств значение соответствующих атрибутов уменьшалось, но не исключалось полностью (в частности, Леда, Европа и т.д.). При этом нужно вспомнить предположение о нечеткой замкнутости мира, "распространяющего грехи отцов на детей до третьего и четвертого поколений". Возникает еще один вопрос, связанный с сопоставимостью категорий, представленных связями AKO. Хорошо спроектированная объектная база данных должна обладать этим свойством. К вопросу согласованности проектного решения мы вернемся несколько позднее.

Очевидно, что для данного примера лучше подходит метод дефаззификации на основе моментов. Это связано с тем, что мы имели дело с обычными свойствами, которые комбинировались для достижения "сбалансированной точки зрения", а не со свойствами, участвующими в процессе принятия решения. При смешении целей в объектной базе данных могут возникнуть проблемы. Необходимо, по крайней мере, изобразить связи AKO (используя окно правил), где для одной объектной базы данных требуются две стратегии.

Таким образом, на нескольких очень простых примерах представлена основная теория нечетких множеств и объяснена ее логика наследования. Внимание было сосредоточено на атрибутах, но наследование применимо и к операциям. Правила в методе SOMA тоже могут быть нечеткими [331]. Рассмотрим более практичный пример.

A.4. Приложение

Рассмотрим проблему, с которой приходится сталкиваться при распределении маркетингового бюджета среди различных сфер деятельности, которые могут привести к увеличению объемов продаж товара. Очевидно, что прибыль от продажи различных типов товаров зависит от распределения бюджета. Рассмотрим следующие методы маркетинга: `Advertising` (реклама), `Promotion` (организация продаж), `Sales Training` (обучение продавцов), `Packaging` (упаковка) и `Direct Mail` (прямая отправка).

Теперь предположим, необходимо сравнить эффективность капиталовложений для увеличения объемов продаж хлебопродуктов к завтраку и пакетов программ. Ясно, что в повышении объемов продаж хлебопродуктов очень большую роль играют реклама, организация торговли и упаковка, но вряд ли эффективна прямая отправка хлебопродуктов потребителям. В такой ситуации распределение ресурсов можно представить с помощью табл. A.1, определяющей долю капиталовложений в процентном соотношении.[4]

Таблица A.1. Процентное соотношение капиталовложений в разные виды маркетинга

	Реклама	Организация продаж	Обучение	Упаковка	Прямая доставка
Хлебопродукты к завтраку	30	50	5	15	0
Пакеты прикладных программ	10	5	15	20	50

Если знание выражено неточно, эти цифры можно легко заменить нечеткими значениями (табл. A.2).

Таблица A.2. Нечеткие данные о процентном соотношении

	Реклама	Организация продаж	Обучение	Упаковка	Прямая доставка
Хлебопродукты к завтраку	около трети	около половины	чуть-чуть	немного	ничего
Пакеты прикладных программ	меньше малого	чуть-чуть	немного	около пятой части	около половины

Здесь приводится два нечетких отношения — для хлебопродуктов к завтраку и пакетов прикладных программ. Эту информацию можно рассматривать как часть общей базы данных товаров. Если ее представить в виде классов, то придется использовать нечеткие объекты и наследование через связи AKO. В качестве примера частичного наследования (рассматриваемых свойств) рассмотрим нечеткий объект, представляющий класс предметов потребления, а

[4] Предупреждение практикующим маркетологам: эти цифры не претендуют на действительно эффективную стратегию.

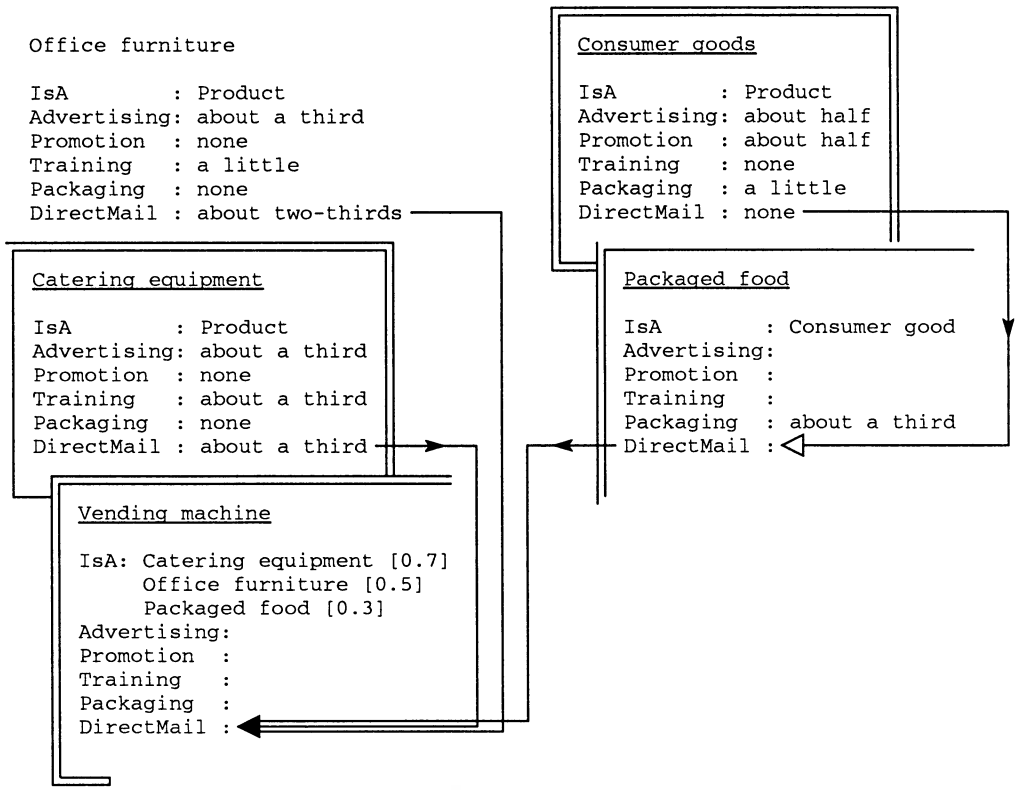
именно торговые автоматы `Vending-Machine`. Автомат можно рассматривать как офисную мебель, оборудование для организаций общественного питания или для упаковки еды, в зависимости от выбранного маркетингового подхода. Нечеткие объекты обеспечивают естественный путь описания этой проблемы.

На рис. А.15 проиллюстрировано частичное наследование нечеткого (т.е. выраженного словами) распределения основных классов товаров. Все свойства являются нечеткими, а неопределенные свойства не содержат значений. Нечеткое множество, возвращаемое атрибуту `DirectMail`, можно дефаззифицировать. Тогда окажется, что для получения наилучших результатов в эту область следует вложить около трети бюджетных средств. Этот результат основан на том факте, что автоматы обладают свойствами оборудования для предприятий общественного питания, офисной мебели и упаковки, о которых многое известно. Действительно, в этом примере дефаззификация по методам моментов и максимума даст приблизительно одинаковые результаты.

Еще одним возможным практическим применением является описание отношений в организациях, где обязанности специалистов в подразделениях организации могут перекрываться или смешиваться в контексте формальной отчетной иерархии. Одно из приложений такой объектной базы — поддержка принятия решений. Другое касается построения формальных моделей, описанных в [626].

Существует огромное число возможностей для применения нечетких объектов. Среди наиболее многообещающих — бизнес-моделирование предприятия и стратегическое планирование. Конечно, можно возразить, что эти задачи можно решить другими методами, но они не обеспечивают такого явного преимущества естественности представления в унифицированном формализме, как нечеткие объекты.

Кратко сравним этот подход с некоторыми другими методами представления неопределенных утверждений об объектах.



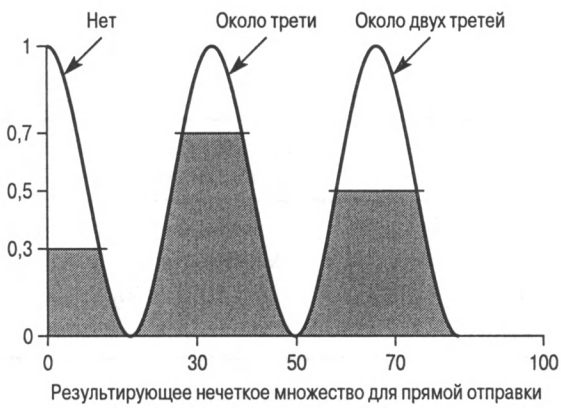


Рис. А.15. Как продавать торговые автоматы

А.5. Нечеткие объекты, нечеткие кванторы и немонотонная логика

Исследуем применение нечетких объектов к одной из классических проблем наследования, используя вместо немонотонной логики нестандартные кванторы. Турецкий обсуждает (и отбрасывает) такой подход, ссылаясь на работу [28]. Но он, вероятно, не знаком с более вдохновляющими результатами Заде по представлению нечетких кванторов и их свойств логического вывода.

Теория Заде позволяет выразить одну из классических мотивационных проблем немонотонной логики в виде: "Большинство птиц умеют летать". Это утверждение может быть четко выражено с помощью нечеткого наследования. Таким образом определяется подход нечетких объектов. Рассмотрим объектную базу данных.

```
Flying-animal (летающее животное)
AKO: Animal
Can-fly: true [fuzz]

Bird (Птица)
AKO: Flying-animal [0.9]
Wings: 2

Penguin (птица)
AKO: Bird
Can-fly: false [fuzz]

Tweety (Певунья)
AKO: Bird [1], Penguin [1]
Can-fly: ?
```

Использованные нечеткие множества проиллюстрированы на рис. А.16. Ответ состоит в том, что Певунья является птицей и не может летать. Аналогичный результат описан в [536]. Пингвины производят движения, напоминающие полет (они машут крыльями, когда ныряют или бегут). Эта информация некоторым образом представлена на рис. А.17.

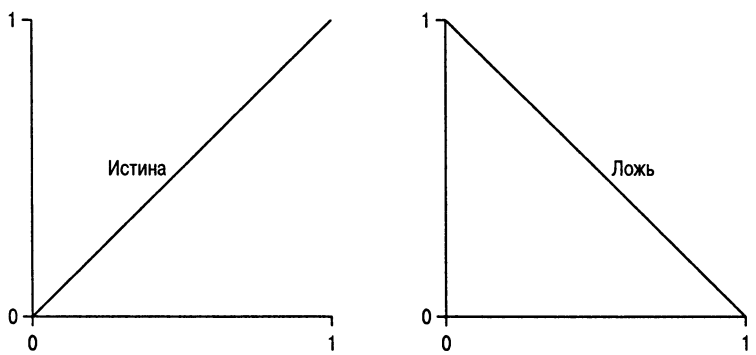


Рис. А.16. Нечеткое множество, для которого значение `true` соотношением задается x=x, а значение `false` — выражением x=-x

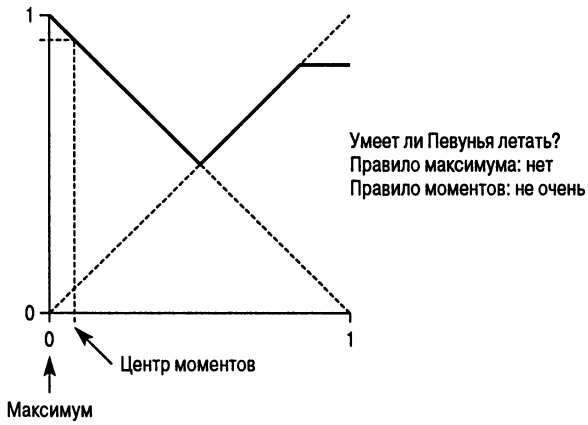


Рис. А.17. Нечеткое множество для утверждения "Певунья умеет летать"

В связи с этим приходит на ум достаточно очевидное обобщение. Числовой параметр, представляющий степень наследования, может быть заменен лингвистической переменной (нечетким множеством или нечетким числом). Это означает, что усечение наследуемых нечетких множеств тоже должно быть нечетким. Такие объекты можно назвать "ультранечеткими" или "дважды нечеткими". Однако поиск формальной семантики становится значительно сложнее, и я подозреваю, что практическое значение такой теории серьезно ограничено ее сложностью. Действительно, это обобщение должно соответствовать точной интерпретации нечетких кванторов, приведенной в разделе А.2.4, где нечеткие кванторы представлены нечеткими числами. Механизм наследования дважды нечетких объектов можно модифицировать, чтобы устранить проблему пересечения/произведения, описанную в разделе А.2.4. Это, конечно, стоит дальнейшего исследования.

Нечеткую AKO-связь полезно интерпретировать как нечеткий квантор типа "большинство/некоторые". Эта интерпретация подходит также для примера с планером. AKO-связи можно использовать (или не использовать) для описания множества противоречивых целей. Хорошая теория проектирования требует интерпретации связей наследования. Альтернативой этому является допущение, что нечеткие объекты имеют фиксированные линии наследования. Тогда наследование можно реализовать через множество различных сетей.

А.6. Бизнес-стратегия и нечеткие модели

Бизнес-стратегия формируется и реализуется в неопределенном мире. Существует множество источников и типов неопределенности. Среди них можно выделить следующие.

- Недостаток понимания.
- Концептуальная ошибка, неопределенность в суждениях, недостаток доказательства, ненадежные источники данных и информации или недостаток конкретности доказательства. Эти причины, наряду с неточностью естественного языка, ведут к изменению степени доверия. Определение может быть словесным, и возможные значения данных

тогда должны будут выбираться из множества лингвистических дескрипторов. Например, в определении клинических симптомов пациента степень его бледности может быть описана только в таких терминах, как "слегка бледный", "ужасно бледный", "жутко бледный" и т.д.

- Противоречащие друг другу или дополняющие друг друга источники фактов.
- Скрытые переменные, вносящие случайный шум.
- Существование исключений.
- Энергия, требуемая для получения определенных данных.
- Случайные события, ведущие к изменению степени правдоподобия.
- Изменение степени возможности или необходимости.
- Инструментальная или экспериментальная ошибка, повреждение сенсорного оборудования, приводящее к изменению точности. Эта ситуация стандартна для научных экспериментов. Ошибки наблюдения обычно распределены по нормальному закону в окрестности истинного значения наблюдаемой переменной.
- Обилие несоответствующих данных, ведущее к изменению степени соответствия.
- Изменение области выполнения предположения.
- Изменение степени истинности или доказуемости предложения.
- Изменение степени дозволенности некоторого действия.
- Изменение степени принуждения или обязательств.

Возможность принятия решения в условиях абсолютной определенности по отношению ко всем соответствующим фактам и рассуждениям — роскошь, редко предоставляемая человеку. Обычно приходится делать предположения относительно данных, которых нет в наличии, о событиях, которые могут произойти или не произойти, и о следствиях, которые могут вытекать из данного решения.

Многие из этих предположений могут быть сделаны несознательно или подсознательно. Некоторые могут быть сделаны точно, с любой степенью достоверности. При формулировке некоторых предположений, сделанных на статистической основе, на помощь приходит математика. В противном случае путеводителем служит практика и опыт.

Традиционно системных аналитиков учат "предпочитать факт мнению" и устранять всю неточность из детального описания. Это хороший принцип, но с ним можно зайти слишком далеко. Принцип несовместимости Заде [825] гласит следующее.

> Когда сложность системы увеличивается, наша способность сделать точные и все еще значимые утверждения о ее поведении уменьшается до тех пор, пока не достигнет порога, за которым точность и значимость (или соответствие) становятся практически взаимно исключающими характеристиками.

Оригинальная мотивация Заде является следствием попыток сконструировать цифровое управляющее устройство для сложного электронного оборудования. Это оказалось сложной задачей, так как по закону необходимости (Law of Requisite) [41] с увеличением сложности

управляющего устройства возрастает и его быстродействие. Оно возрастает настолько, что цифровые управляющие устройства становятся либо неэффективными, либо настолько сложными, что их перестают понимать даже сами конструкторы.

Один из способов преодоления противоречия — представить неопределенное или неточное знание человека прямо, а не с помощью некоторого искусственного представления, такого как точная формула. В контексте автоматического управления оператор формулирует стратегию управления следующим образом: "Когда прибор начинает перегреваться, я должен повернуть эту ручку вниз немного больше", а не "Когда температура возрастает выше 98,4, входной рычаг поворачивается на 4,7%". Этот подход используется в теории нечетких множеств. Он может быть применен как в стратегическом планировании, так и в промышленных цифровых управляющих устройствах.

Ниже приводится пример стратегии, принятой менеджером по продажам товаров массового потребления, например чистящих порошков. Аналогичный пример уже использовался в главе 5. Он описан кратко, но представляет достаточно четкую картину того, как цены устанавливаются на практике.

1. Наша цена должна быть низкой
2. Наша цена должна вдвое превышать себестоимость
3. Если цена конкурентов не очень высока, то наша цена должна быть близкой к цене конкурентов

Существует несколько моментов, которые следует отметить в связи с этой стратегией. Данная стратегия выражается в неопределенных терминах и все же хорошо понятна каждому менеджеру по продажам и воспринимается как руководство к действию. Примечательно, что описание этой программы на английском языке на самом деле представляет собой программный код, написанный на языке REVEAL.

```
1. OUR PRICE SHOULD BE LOW
2. OUR PRICE SHOULD BE ABOUT 2*DIRECT.COSTS
3. IF THE OPPOSITION.PRICE IS NOT VERY HIGH THEN OUR PRICE
SHOULD BE NEAR THE OPPOSITION.PRICE
```

Дело в том, что REVEAL для реализации лингвистических термов использует нечеткие множества, такие как LOW. Рассмотрим, как выполняется эта программа.

Сначала рассмотрим два нечетких множества, LOW и HIGH. Их можно определить как векторы по шкале цен для чистящего порошка (рис. А.18, *а* и *б*). VERY — это оператор, который вычисляет квадрат значения каждой точки кривой, представляющей нечеткое множество. Результирующее множество VERY HIGH изображено на рис. А.18, *б*. Слова OUR, SHOULD и THE рассматриваются как шум. Слово BE является синонимом для IS и NEAR и означает то же, что и ABOUT.

Первое утверждение стратегии означает, что цена должна быть как можно более совместимой с LOW, т.е. цена должна быть равна нулю. Это противоречит предположению о том, что она должна вдвое превышать себестоимость. Примечательно, что нечеткая стратегия автоматически разрешает это противоречие, выбирая ту цену, которая дает максимальное значение истинности для пересечения нечетких множеств (рис. А.18, *г*). Пик области пересечения представляет гибкое ограничение, или область достижимости для цены. На рис. А.18, *в* изображено нечеткое множество ABOUT 2*DIRECT.COSTS.

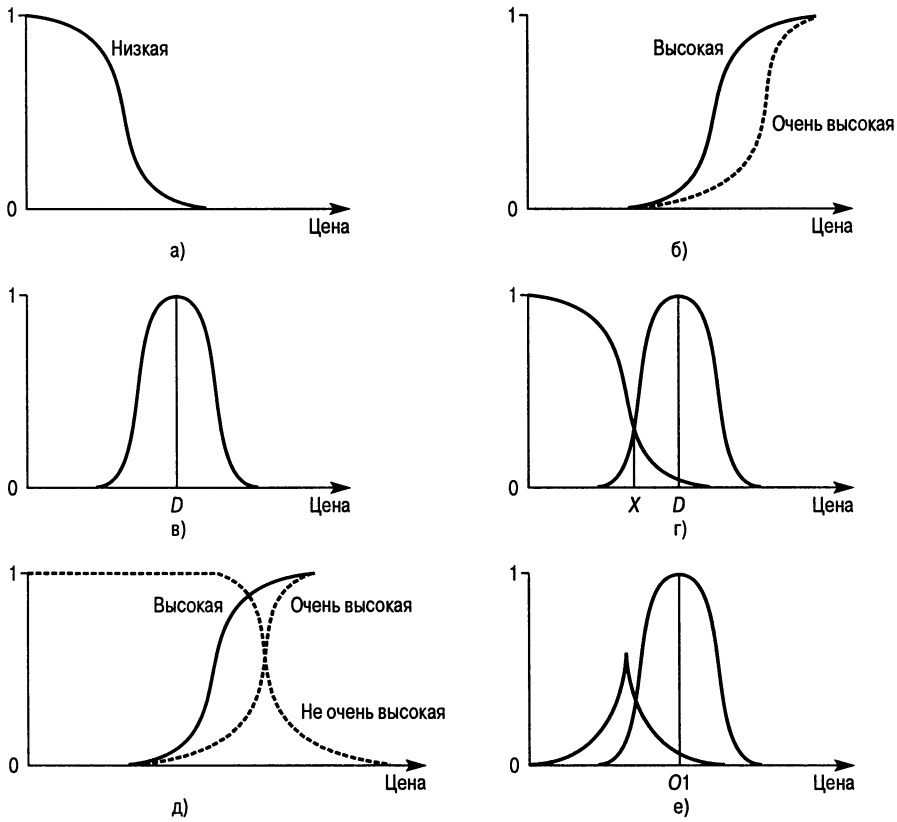


Рис. А.18. `LOW` (а), `HIGH` (б) и некоторые производные нечеткие множества: `ABOUT D` (в), `LOW AND ABOUT D` (г), `NOT VERY HIGH =1- HIGH^2` (д), `(LOW AND ABOUT D) AND NEAR O1` (е)

Теперь следует интерпретировать правило 3. Берем реальное значение для конкурентов `O` и вычисляем для него значение истинности `NOT VERY HIGH`. Оно равно *T*. Нечеткое правило вывода интерпретируется как усечение выходного нечеткого множества `NEAR OPPOZITION` на уровне *T*. В результате находим объединение множества `LOW AND ABOUT D` с этим усеченным множеством. Здесь `D` обозначает `2*DIRECT.COST`. И наконец, если необходимо получить реальное значение цены, а не нечеткое множество, нужно выполнить дефаззификацию. В этом случае выбираем метод максимума. На рис. А.19, *б* и *в* показано, что здесь возможно два случая. Если значение `T` превышает максимальную степень истинности в области достижимости, получаем резкий скачок на выходе от `R2` к `R1`. Это соответствует реальной жизненной ситуации: выходное решение является разрывным. В процессе управления требуется гладкое решение, поэтому следует использовать правило дефаззификации на основе центра моментов.

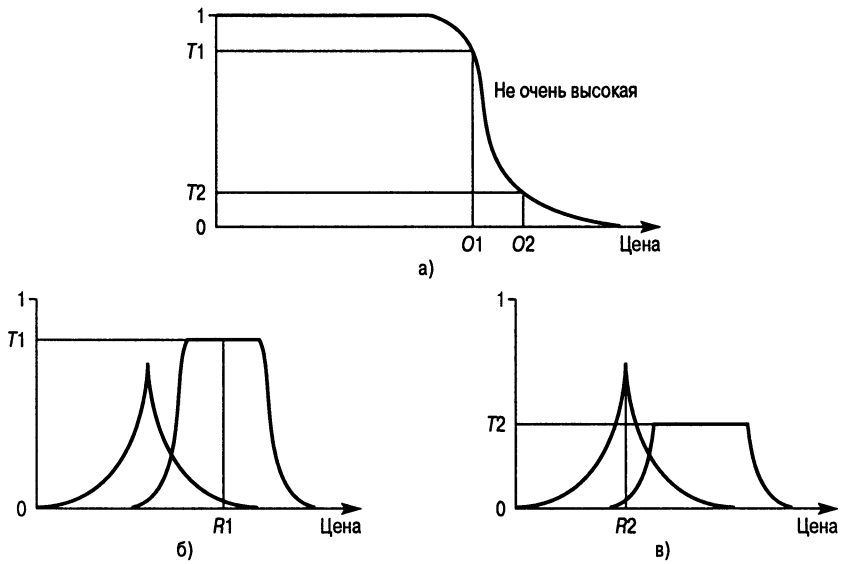


Рис. А.19. Нечеткий вывод: а — Tj представляет степень истинности утверждения `NOT VERY HIGH` для Oi; б — случай 1: Ti>max A; в — случай 2:Tj<max A, `A=LOW AND ABOUT D`

Цель этого примера — показать, как нечеткие правила, используемые для моделирования бизнес-стратегии, могут быть применены для обеспечения достаточно точной, хотя возможно нелинейной и сложной, модели поведения. В SOMA множества правил встроены в объекты и могут наследоваться. Таким образом, бизнес-стратегию можно моделировать и с помощью множества нечетких правил. Можно либо создать специальный объект целевой стратегии, на который ссылаются все другие объекты, либо распределить стратегию по соответствующим объектам. Конкретная реализация зависит от приложения.

В SOMA не обязательно использовать нечеткие правила и даже правила вообще. Однако они обеспечивают очень мощную технологию моделирования, которая помогает сформировать общее понимание проблемы пользователями и разработчиками.

А.7. Правила управления для нечетких систем множественного наследования

Подытожим возможные режимы управления с помощью теории нечетких объектов. Если они формулируются с помощью окна правил наиболее общего объекта, то являются глобальными правилами. В противном случае они могут представлять собой локальные вариации.

- **Режим по умолчанию**. Если атрибут имеет значение, не наследуйте его. Если нет, то объединяйте усеченные наследуемые атрибуты с помощью оператора взятия максимума. Для обычных переменных максимум факторов уверенности представляется

значениями АКО и связывается с наследуемым значением. Множественное наследование четких значений может привести к получению многозначных атрибутов (списков).

При этом могут использоваться различные нечеткие логики. Оператор максимума может быть впоследствии заменен соответствующей t-конормой. Для отдельных АКО-атрибутов оператор максимума (или конормы) может быть заменен оператором минимума (или соответствующей нормы). Максимум выбирается в случае конфликтующих значений для альтернативных точек зрения. Выбор минимума соответствует максимальной предосторожности и предположению о том, что конфликтующие источники дополняют друг друга.

- **Нечеткое предположение о замкнутости мира**. Этот подход предполагает наследование и объединение всех определенных значений. Его можно несколько ослабить. Стратегия управления не допускает исключений, но иногда это именно то, что нужно. Например, когда мы делаем вывод, что собака имеет четыре ноги, потому что она млекопитающее, а человек является исключением — имеет только две, мы решительно предпочитаем наивную физику зрелой биологии. Человек (обычно) действительно имеет четыре ноги, просто две из них были приспособлены для других целей. С такой точки зрения необходим механизм наследования, который передает по наследству свойства млекопитающих несмотря на человеческую исключительность.

Каждый из режимов делится на два вида (по числу методов дефаззификации). Это предоставляет создателю системы выбор, по крайней мере, из 24 режимов управления.

Таким образом, ряд допустимых режимов управления определяется ответами на следующие вопросы.

1. Следует ли наследовать определенные атрибуты?
2. Как реализовать множественное наследование: на основе объединения или пересечения?
3. Какую из нечетких логик выбрать?
4. Какой из методов дефаззификации следует использовать?

А.8. Теория проектирования нечетких объектов

В этом разделе исследуются некоторые задачи, возникающие при построении нечетких объектных систем, и предлагаются некоторые рекомендации.

Общие проблемы множественного наследования атрибутов или свойств возникают не только при работе с нечеткими множествами. В [761] перечислены аналогичные проблемы для четких систем и предлагаются некоторые пути их решения. Идею иерархического упорядочения расстояния Турецкого, на мой взгляд, можно использовать для реализации наследования нечетких объектов.

В связи с этим в качестве вводного замечания стоит отметить, что в семантике Турецкого таблицы истинности генерируются на основе трехзначной логики Клина (Kleene), а не логики Лукашевича (Lukasiewitz) [296]. Преимущество логики Лукашевича — условные утверждения (о будущем) для неопределенного терма и неопределенные значения. Аргументом в пользу системы Турецкого является наличие связей, указывающих на невозможность получения

заключения (или наследования значения). Разница между моим подходом и методологией Турецкого состоит в следующем: конфликт в наследовании нужно интерпретировать не как неразрешимость, а как возможность или потенциальность.

A.8.1. Полнота проектного решения

Предлагаемый автором подход имеет некоторые преимущества. И это не просто теоретические рассуждения. Есть некоторые проблемы, которые подход Турецкого не позволяет решить. Вывод о том, что объект `Hang-glider` безопасен, потому что это игрушка, может рассматриваться как один из способов разрешения конфликта. Можно было бы разорвать связь, используя дополнительный класс объектов `Dangerous-toy` (Опасные модели). Но этого не делается, а применяется подход упорядочения расстояний. Однако при этом в объектную базу должны быть внесены все возможные варианты декомпозиции. В противном случае небольшое изменение в проектном решении может привести к абсолютно отличному поведению при наследовании. Поэтому требуется процедура, определяющая полноту объектной базы. Необходимо развивать теорию проектирования для общих объектных баз данных. Это может оказаться очень сложной проблемой. Отсутствие такой теории вынуждает использовать некоторый метод обоснования по умолчанию, основанный на предположении Рейтера (Reiter) о замкнутости мира (а не упомянутую выше его нечеткую версию).

Аналогичный вопрос о полноте возникает и при попытке структурировать знания и данные. Даже в традиционных моделях данных сущность-связь, согласованный проект достигается только большим мастерством. На помощь приходит теория нормальных форм, но в реальных динамических системах полноту очень сложно обосновать.

Более того, поскольку опасная игрушка — это случайный объект, о котором речь шла в разделе 9.2, то это должно послужить тревожным сигналом.

A.8.2. Объекты или атрибуты

Свойства объектов предметной области можно определить через объекты или атрибуты. С выбором способа реализации свойств связана основная проблема проектирования. Рассмотрим два эквивалентных объекта.

Объект-1 <<fuzzy>>
AKO: ? Свойство-A: степень [fuzz]

Объект-2 <<fuzzy>>
AKO: Объект-со-свойством-A [степень]

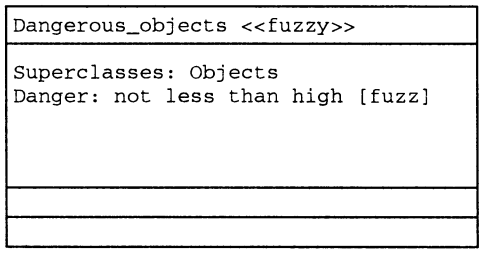
Проблема состоит в выборе между альтернативными формулировками. Будем называть такие объекты **тавтологическими**.

В связи с этим возникает вопрос о статусе двух типов нечеткости: нечеткости связи между классами (или представителями классов) и нечеткости, наследуемой в предикатах описания (атрибутах). Необходимо понять, можно ли смешивать различные типы неопределенностей в контексте грамотного проектирования. С одной стороны, следует придерживаться стройности

проектного решения, но с другой стороны, если позволить программисту смешивать обе формы, получаются более выразительные результаты.

А.8.3. Тавтологические объекты и максимальная декомпозиция

Как же охарактеризовать тавтологические объекты?



Тавтологические объекты содержат только один не-АКО-атрибут со значением предиката и именем, соответствующим имени объекта. АКО-атрибут не привносит никакой новой информации. Согласно спецификации, опасный объект — это объект, для которого степень опасности принимает значение "высокая" или более.

В четких системах следовало бы автоматически не допускать таких вариантов. Но в приведенном выше примере планера это неизбежно. Иначе никак нельзя задать минимальное значение степени безопасности объекта. Это согласуется с интуицией, так как действительно не существует *априорной* причины (по крайней мере, в пределах данной объектной базы), по которой следует считать планер опасным, но он на самом деле **является** опасным. Это удобный путь разрешения проблемы, которая возникает и при создании четких объектных систем. С другой стороны, иногда такие объекты выглядят неестественно. Например, слон — серый, а серый объект является однотонным. И снова вопрос существенного и случайного выступает на первый план.

Одна из интересных точек зрения на эту проблему состоит в том, что наличие тавтологического объекта в проектном решении предполагает наличие логических структур, подобных решеткам Келли (Kelly). Поэтому тавтологические объекты в проектном решении обеспечивают возможность использования методов извлечения знания, вытекающих из решеток Келли.

Рассмотрим еще одну проблему, возникающую в теории неизбыточных (или нормальных) форм для объектных баз данных, — циклы. Для этого чуть ниже рассмотрим соответствующий пример. При использовании четких объектов эта проблема не возникает, поскольку приведенный ниже цикл всегда представляет равенство. В случае нечетких объектов сложнее определить, существует ли необходимость для таких отношений.

Например, фраза *Большинство людей являются жадными* может быть представлена с помощью нечетких объектов в следующем виде.

Men <<fuzzy>>
AKO: Greedy-men [0.8]

Greedy.men <<fuzzy>>
AKO: Men [1]

Эта пара нечетких объектов содержит неприводимый цикл. Системы четкого наследования обычно требуют ацикличности. Это, конечно, удобно с вычислительной точки зрения, но, теоретически, совсем не обязательно. Однако при использовании нечетких объектов циклы не сворачиваются. Один из путей устранить проблему — это не допускать циклов, а создать новый объект, представляющий "скупую сущность" (например, автомобиль, "пожирающий" бензин). Тогда жадный человек `Greedy.men` является алчным со степенью достоверности `[1]` и человеком со степенью доверия `[1]`, а человек скуп со степенью `[0,8]`. Циклы при этом исключаются. В процессе этих рассуждений было нарушено правило отсутствия тавтологий.

Тесная связь между ацикличностью и наличием тавтологических объектов приводит к следующим правилам проектирования.

- Тавтологические объекты следует вводить для предотвращения циклов.
- Тавтологические объекты можно использовать, чтобы избежать явного задания свойств объекта пользователем. Однако это приводит к проблеме, связанной с наследованием двумя объектами одного и того же значения атрибута от тавтологического объекта.
- В остальных случаях тавтологических объектов следует избегать.

На этом завершим рассмотрение теории проектирования нечетких объектов и перейдем к вопросам представления знаний в целом.

A.9. Связь нечетких объектов с другими понятиями

В этом заключительном разделе указывается, что нечеткие объекты являются естественным обобщением как отношений, так и объектов. Это утверждение является неформальным, но имеет свое математическое обоснование. Этот раздел можно пропустить без потери смысла всего предыдущего изложения.

А.9.1. НЕЧЕТКИЕ ОБЪЕКТЫ КАК ОБОБЩЕНИЕ НЕЧЕТКИХ ОТНОШЕНИЙ

Первое наблюдение, которое следует сделать, состоит в том, что нечеткие отношения обобщают отношения вообще.

Отношение — это подмножество некоторого декартова произведения множеств. Его можно считать функцией, отображающей это произведение во множество истинности, содержащее в классической логике два значения {0; 1}. Нечеткое отношение является такой функцией, для которой образ представляет собой решетку истинности многозначной логики. В рассматриваемом случае эту роль выполняет единичный интервал. Возможно также табличное представление.

Хорошо известно (по крайней мере, в математике), что существует взаимно однозначное соответствие между классами функций $A \times B \rightarrow I$ и $A \rightarrow I^B$. Это также имеет место для n-мерных декартовых произведений. Для получения этого результата каждой функции $f(a,b)$ ставится в соответствие функция, отображающая точку a в нечеткое множество

$$g{:}B \rightarrow I{:}\ b \rightarrow f(a,b).$$

Это значит, что в табличном представлении форма

Любит:	Человек 1	Человек 2	Степень
	Джон	Мери	0,9
	Мери	Джон	0,2
	Джил	Мери	0,4

соответствует форме

Любит:	Человек	Распределение возможностей
	Джон	π1
	Джил	π2
	Мери	π3

Распределение возможностей задают нечеткие множества (не изображенные здесь). Например, π1 означает степень любви Джона к каждому другому человеку в пространстве суждений. Значение истинности этой величины для Мери равно 0,9.

Таким образом, синтаксис нечетких объектов (не считая атрибутов АКО) соответствует синтаксису, принятому для нечетких отношений.

Следует отметить, что отношения могут образовывать категории, которые отражают некоторые желаемые свойства отношений, такие как проекции или объединения. Из принципа расширения Заде [432] следует, что нечеткие отношения также могут образовывать категории. Очевидно, что функтор соответствия отношений и нечетких отношений имеет вид

$$(Rel) \rightarrow (FuzRel).$$

Это подтверждает, что нечеткие отношения (базы данных нечетких отношений) обобщают классические отношения (базы данных отношений).

Если к нечетким отношениям добавить структуру наследования, окажется, что нечеткие объекты (классы) обобщают нечеткие отношения.

A.9.2. Нечеткие объекты как обобщение объектов

Аналогично класс (как набор объектов) также обобщает отношения.

Ключевое отличие между объектами и отношениями состоит в том, что атрибуты объектов могут иметь не только атомарные значения. В качестве атрибутов могут выступать податрибуты (фацеты), методы или списки, и (в следствие наследования) число атрибутов в определении объекта нельзя предсказать наперед. Это означает, что в основе нечетких объектов лежит логика, которая не является логикой первого порядка (как с нечеткими отношениями). Кроме того, мы работаем в потенциально бесконечном (счетном) пространстве декартова произведения. Интуитивно это означает, что объекту может быть присвоен *любой* атрибут: голубизна, голод, острота и т.д. Реальные объекты обладают лишь несколькими соответствующими свойствами, а модельные объекты могут наследовать что угодно. Экземпляры являются исключением из этого правила только при использовании одиночного наследования. Они наследуют свойства *всех* родительских классов, а значит, потенциально могут принадлежать ко всем классам.

Класс представляет собой множество объектов. Каждый кортеж в отношении соответствует одному объекту класса. Конечно, экземпляры класса сами по себе могут быть многозначными.

Считая классы абстрактными типами данных, рассмотрим их как алгебры. Таким образом, могут существовать отображения (методы), сохраняющие алгебраическую структуру. С этой позиции множества объектов или классы могут образовывать категории. Таким образом, снова получим функтор включения

$$(Rel) \rightarrow (Class),$$

который связывает отношения с объектами.

Практическое применение этой идеи состоит в том, что системы поддержки истинности и системы временной логики приобретают очень согласованный каркас реализации, которому соответствуют многие методы теории реляционных баз данных.

Поскольку атрибуты нечетких объектов могут представлять собой нечеткие множества, то нечеткие объекты можно рассматривать как обобщение традиционных объектов.

Здесь изложена только основная идея, которую еще предстоит исследовать.

A.9.3. Нечеткие объекты как универсальное обобщение

Возникает вопрос, *можно ли объединить два приведенных выше функтора*. Если мы сможем построить это объединение, то будем иметь универсальное обобщение как объектов, так и нечетких отношений. Несложно предположить, что очень обоснованными претендентами на роль объединения являются нечеткие объекты. А значит, нечеткие объекты обобщают все представления объектного знания.

Если удастся доказать этот результат или даже значительно более слабую его версию, представленную на рис. A.20, то будет сделан шаг к унифицированной теории, покрывающей следующие вопросы представления знаний.

- Реляционная модель данных.
- Объектная ориентация и системы наследования.
- Нечеткие отношения и извлечение нечеткой информации.

- Немонотонные рассуждения.
- Временные рассуждения, возможные миры и системы модальных логик.

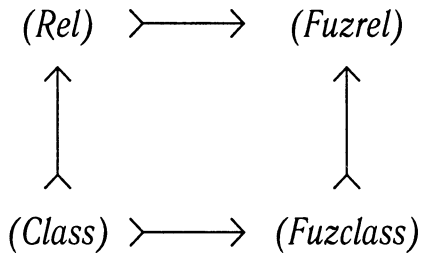


Рис. А.20. Предположительное обобщение в категории категорий

Это смелое заявление. Приведенные выше аргументы являются экспериментальными. Однако, в конце концов, поставленная цель стоит дальнейших исследований. Я надеюсь, что ученые выберут некоторые из этих идей в качестве направления для изучения.

А.10. Резюме

В этом приложении неформально определены синтаксис и семантика нечетких объектов и соответствующего нечеткого расширения метода SOMA. Изложенный подход легко можно включить в стандартные системы обозначений, такие как UML. Автор попытался показать, как текущий синтаксис UML можно использовать для описания нечетких объектов.

Нечеткие объекты в SOMA могут иметь нечеткие значения атрибутов. Здесь была описана теория множественного наследования для нечетких объектов. Нечеткие объекты могут наследовать как четкие, так и нечеткие значения атрибутов *частично*, т.е. с заданным значением фактора определенности. В приложении показано, как эти средства применяются при моделировании частичного наследования свойств. Было рассмотрено также частичное наследование методов на основе фактора определенности. В качестве правил можно выбирать нечеткие правила.

Было предложено несколько возможных приложений. Нечеткие объекты могут оказать огромную пользу при моделировании деятельности предприятий и реорганизации бизнес-процессов. Одним из наиболее сложных (и слабо проработанных) приложений нечетких объектов являются системы интерпретации утверждений естественного языка. Хочется надеяться, что ученые подхватят эту идею в качестве темы для исследования. Мой интерес сосредоточен в практических сферах. Автор также предложил направление исследования для математиков, интересующихся теорией баз данных и абстрактной алгеброй отношений. Я надеюсь, что эта задача также будет подхвачена и решена академическим сообществом.

Были также исследованы методологические вопросы проектирования нечетких объектов. Предложено множество нерешенных проблем в этой области в качестве тем для исследования. Вообще, я надеюсь, что изложенная здесь теория будет подтверждена конкретными приложениями. Существует очевидный резонанс между затронутыми здесь вопросами и текущими проблемами в объектно-ориентированных базах данных.

Изложенная здесь теория готова к применению. Это дает возможность переходить к стадии реализации.

A.11. Дополнительная литература

Первая версия этой теории описана в [330], а более поздняя в [331]. Последняя работа является хорошим путеводителем по обширной литературе по теории нечетких множеств и систем. Классической книгой по этому вопросу считается [236]. Работа [432] является известным введением в базовые математические методы, а книга [331] более ориентирована на приложения и охватывает теорию нечетких систем на основе правил, включая систему REVEAL. Журналы *Fuzzy Sets and Systems* и *The Journal of Approximate Reasoning* содержат новейшие результаты как теоретических, так и прикладных исследований.

Другие, несколько отличные определения нечетких объектов или нечеткого наследования можно найти в работах [50, 200, 237, 590, 768].

В последнее время количество публикаций на эту тему значительно уменьшилось, но [621] представляет собой интересный и достаточно оригинальный вклад в эту проблему. В [498] описано понятие избирательного наследования для нечетких объектов. В [201] приведено иное определение нечеткого объекта, несколько напоминающее одно из приведенных в настоящей главе, и показано, как его можно применить к географическим информационным системам.

Литература по фреймовым системам AI также достаточно обширна. В [801] и [483] излагаются теоретические вопросы, а [697] представляет собой информативное введение в проблему.

Б

Ранние методы анализа и проектирования

Принцип, в соответствии с которым сохраняется любая незначительная, но полезная мутация, я назвал Естественным отбором. Давайте рассмотрим борьбу за существование более детально.

Чарльз Дарвин. *Происхождение видов*

Это приложение предназначено для тех, кто интересуется происхождением современных методов моделирования требований, объектно-ориентированного анализа и проектирования. В нем приводится обзор приемов моделирования, используемых до появления языка UML, а также описываются результаты многочисленных исследователей, которые существенно повлияли на современные методы, а в некоторых случаях до сих пор продолжают определять развитие этого направления.

Б.1. Ранние методы и системы обозначений, используемые при объектно-ориентированном проектировании

Возможно, среди множества объектно-ориентированных методов проектирования самый первый был предложен Гради Бучем (Grady Booch). В его первоначальных статьях [93, 94] содержалось четкое изложение метода проектирования для языка Ada, однако при этом вводился новый термин. С этой точки зрения объектно-ориентированное проектирование базируется скорее на принципе инкапсуляции, а не на композиции функций (как при использовании традиционных методов) или на полностью объектно-ориентированном подходе с применением наследования. Такую инкапсуляцию можно легко реализовать с использованием пакетов языка Ada или модулей Modula-2, тогда как наследование и полиморфизм в этих иерархически структурированных языках представляют собой неестественные конструкции. Однако передачу сообщений можно без особого труда смоделировать с помощью вызовов функций или процедур и тем самым реализовать конструктор и операции доступа к объекту. Позднее Буч существенно расширил и усовершенствовал свой метод, который уже не был столь тесно связан с языком Ada. Данный подход значительно повлиял на язык UML. Этот вопрос отдельно будет обсуждаться ниже.

Booch86

Первоначальный метод Буча начинается с анализа потоков данных, результаты которого затем используются для поиска конкретных и абстрактных объектов в пространстве предметной области. На диаграмме потоков данных (DFD — Data Flow Diagram) они изображаются в виде кружков и хранилищ данных. Далее по кружкам, обозначающим на той же диаграмме процессы, идентифицируются методы. Альтернативным и, вместе с тем, дополняющим подходом является текстовый анализ [2], который описывался в главе 6. Применение этого подхода можно автоматизировать точно так же, как и использование некоторых средств поддержки метода HOOD и подхода [680]. В большинстве методов, основанных на работе Буча, использовалась некоторая форма текстового анализа. В системе обозначений Буча объекты представляются в виде пятен или "облаков" (рис. Б.1). Стрелки указывают, какие объекты используют службы (или методы) других объектов. Тем самым определяются отношения клиент/сервер (или использование) и обработка сообщений.

Рис. Б.1. Изображение отношения клиент/сервер между объектами с использованием системы обозначений Буча

Группы объектов разделяются на "подсистемы" управляемого размера, которые соответствуют уровням или слоям на диаграмме DFD (рис. Б.2, *а*). Классы соответствуют пакетам Ada (рис. Б.2, *б*). В своей системе обозначений Буч не использует изображения атрибутов,

поскольку весь доступ к интерфейсу объекта выполняется через метод или операцию, хотя сам язык Ada разрешает обращаться к структурам данных внутри описания пакета. Затененная область представляет реализацию или тело пакета, а незатененный прямоугольник (или параллелограмм для параметризованных классов) — описание. Незатененные овалы задают имя пакета или объявление типа, а каждый из незатененных прямоугольников — метод.

Показанные на рис. Б.2, *б* и Б.3 условные обозначения известны как градиграммы (Gradygram). Они существенно повлияли на представление модулей в языке UML. Между объектами Буч выделяет два типа отношения клиент/сервер (рис. Б.3). Если стрелка начинается от затененного прямоугольника, представляющего тело объекта A, то реализация этого объекта зависит от служб, предоставляемых объектом B. Если же стрелка начинается от обозначения объявления типа, то от объекта B зависит только описание (или интерфейс) объекта A. Все это скорее относится к проектированию, а не к анализу.

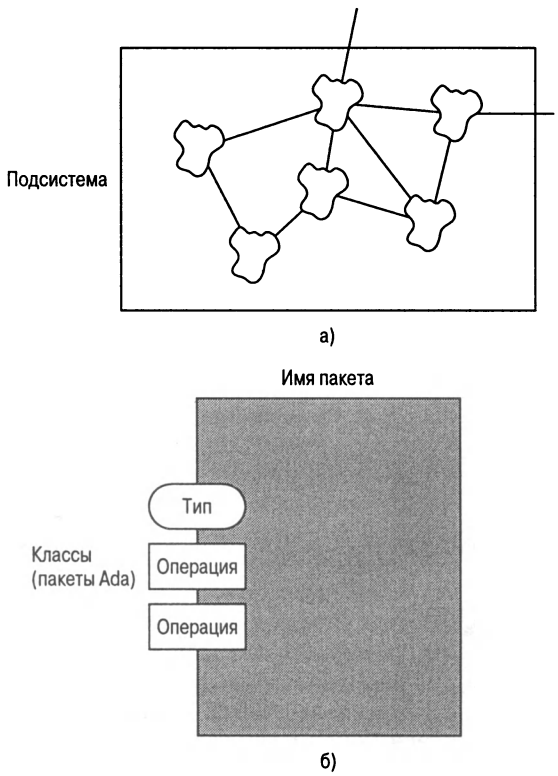


Рис. Б.2. Уровни в системе обозначений Booch86 (а). В нотации Booch86 классы отождествляются с пакетами; при этом указываются только тип и методы (б)

В [723] язык Ada рассматривается как универсальный язык проектирования. Такой подход является стандартным для разработчиков генераторов кода на основе методов, которые на промежуточной стадии перехода от графического или текстового проектирования к реальному

коду используют Ada-подобный язык описания программ (PDL — Program Description Language). В языке Ada классы отождествляются с абстрактными типами данных и соответствуют пакетам, которые (на жаргоне программистов Ada) экспортируют частично закрытые или закрытые типы. Экземпляры соответствуют экземплярам закрытых или частично закрытых типов или пакетов, которые играют роль абстрактных конечных автоматов. Методы соответствуют подпрограммам, экспортируемым из описания пакета. Практический опыт показывает, что если необходим единственный экземпляр класса, то он должен быть определен как пакет, однако если необходимо несколько экземпляров, тогда следует определить класс. Если к состоянию объекта должны иметь доступ другие объекты, такой объект должен определяться в результате инстанцирования некоторого класса. Наследование в языке Ada реализовать достаточно трудно, однако в некотором ограниченном виде это можно осуществить с использованием производных типов или родовых пакетов.

С момента выхода первой работы Буча предложенный им метод не остался без изменений. В более поздней работе [98] показаны расширенные перспективы применения объектно-ориентированного проектирования, которое уже не было так сильно связано ни с языком Ada, ни с любым другим языком программирования. Пересмотренную систему обозначений и методику можно использовать при программировании на языках Smalltalk, CLOS, C++, Ada, а также большинстве других объектно-ориентированных или основанных на объектах языках. Кроме того, Буч сформулировал достаточно общие рекомендации по выявлению объектов и управлению объектно-ориентированным проектированием и программированием. Этот модифицированный метод более подробно будет рассмотрен ниже в отдельном разделе.

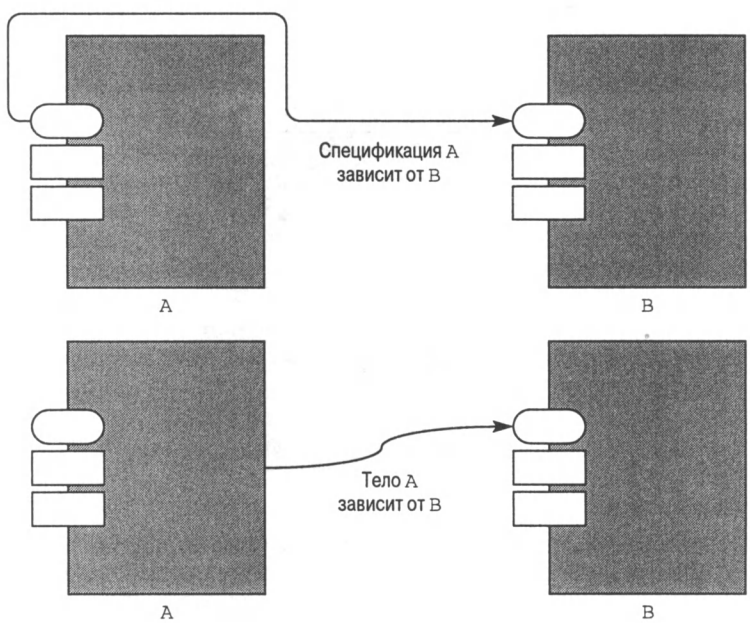


Рис. Б.3. В нотации Booch86 клиенты, использующие описание или интерфейс серверов, отделены от клиентов, использующих их реализацию

Новаторская работа Буча послужила причиной появления иных методов проектирования, тесно связанных с разработкой систем на языке Ada. К таким методам относятся GOOD, HOOD и MOOD. Ниже будут рассмотрены два из этих методов. Некоторые другие методы проектирования и системы условных обозначений, которые не связаны с языком Ada, по существу, базируются на тех же основных принципах. Достаточно подробно будут рассмотрены два таких метода: метод OOSD и подход, описанный в работе [98]. Другие три методики, OODLE, JSD и методология анализа границ/проектирования (RDD/CRC), будут описаны лишь вкратце. Среди иных методов и систем обозначений, которые не обсуждаются в данной книге, следует упомянуть метод, широко используемый после появления работы [129]. Этот подход был использован при разработке CASE-средства для коллективной разработки на языке Ada, в методе Texel, а также оказал влияние на такие методы, как OODLE.

GOOD

Метод GOOD (General Object-Oriented Design) был разработан специалистами NASA Сейдевицем и Старком [693]. Он предназначен как для разработки спецификации требований, так и для проектирования в рамках проектов с использованием языка Ada. Подобно методу Буча, в рамках метода GOOD предварительно создается многоуровневый набор диаграмм потоков данных, на основе которых идентифицируются объекты. На диаграммах DFD выявляются внешние взаимосвязи, хранилища данных, управляющие интерфейсы и хранилища управляющей информации. При этом классы определяются путем исследования потоков данных и потоков управления. Изучение основных процессов позволяет сформировать абстрактную модель функционирования системы, а отслеживание входящих и исходящих потоков, связанных с этими процессами, позволяет построить набор многоуровневых диаграмм. На этих диаграммах основное внимание акцентируется на отношениях управления и передачи данных между сущностями. Таким образом, сущности и группы сущностей становятся объектами, а выделенные операции преобразования данных соответствуют их методам. Как и метод Буча, а также другие методы, основанные на использовании языка Ada, подход GOOD предусматривает формирование нисходящей иерархии объектов, которая основывается на том, как объекты используют друг друга.

HOOD

Идея иерархии старшинства используется также и в другом методе — HOOD. В названии метода буква *H* означает *иерархический* (на английском *hierarchical*). Нетрудно догадаться, что аббревиатура *OOD* (Object Oriented Design) означает *ООП* (объектно-ориентированное проектирование). Метод HOOD был разработан Европейским космическим агентством (European Space Agency). В основном, он предназначался для разработки программ на языке Ada. На этот подход непосредственное влияние оказал метод GOOD, и, кроме того, многое было позаимствовано из метода Abstract Machines компании Matra Espace и методик компании CISI Ingeniere (обе зарегистрированы во Франции). Как и в рамках метода GOOD, в методе HOOD основное внимание акцентируется на иерархиях композиции ("целое-часть"), однако при этом не рассматриваются иерархии классификации (наследования). В методе HOOD объекты могут быть либо пассивными, либо активными. Пассивные объекты могут использовать лишь службы других пассивных объектов, а активные объекты — службы любых объектов. HOOD является методом нисходящей декомпозиции, который позволяет выполнять декомпозицию объекта верхнего уровня, а затем производить дальнейшее разбиение

полученных объектов. По существу, в методе HOOD существует две иерархии: композитная и иерархия использования. Иерархию использования можно рассматривать как сеть.

В рамках метода HOOD при построении диаграмм используется стиль Буча, т.е. градиграммы (Gradygram). Типичная система обозначений представлена на рис. Б.4, где также показано, как с использованием специальной пиктограммы следует обозначать отношения включения (т.е. формировать композитную иерархию). Отношения старшинства или использования изображаются с помощью стрелок, направленных в сторону объекта-потомка (рис. Б.5, *а*). Операция родительского объекта `start` (рис. Б.5, *б*) реализуется при помощи операции `begin` дочернего объекта A, которая в полном объеме выполняет все требуемые функции. Связь "реализуется" — это чрезвычайно полезный прием декомпозиции, который существенно повлиял на метод SOMA.

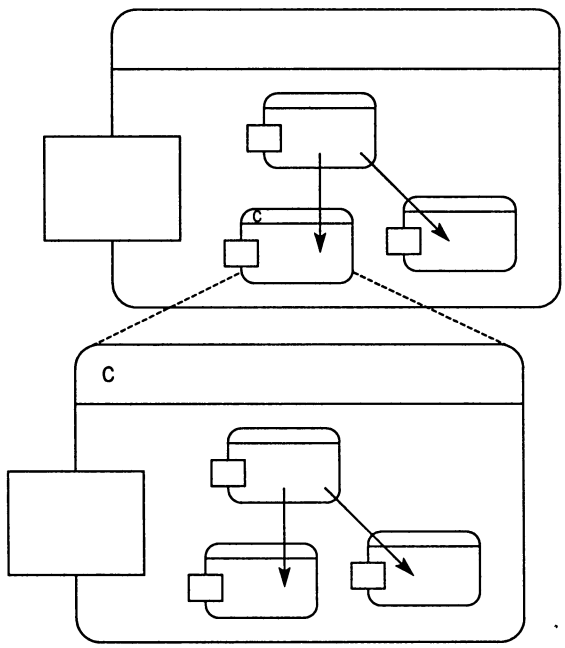


Рис. Б.4. Объектная декомпозиция с использованием метода HOOD и отношений использования и включения. При этом стрелки интерпретируются как отношения использования, а внешний контур пакета на диаграмме свидетельствует о том, что данный пакет является частью другого пакета

В методе HOOD декомпозиция каждого объекта выполняется за несколько шагов, первый из которых связан с базовым проектированием. На этом этапе применяются различные приемы построения диаграмм, характерные для других методов структурного анализа и проектирования. Полученные в результате понятия изображаются в виде объектов и внешних интерфейсов. На этом этапе выполняются различные виды деятельности, в частности формулировка проблемы (обычно в текстовом виде), а затем анализ и структурирование данных. При этом поддерживается упоминавшаяся выше идея Эбботта (Abbott), предполагающая

использование существительных и глаголов для идентификации объектов и методов. В качестве примера идентификации объектов с помощью традиционных подходов рассмотрим, как понятия можно преобразовать при помощи обычного метода структурного проектирования Йордана (Yourdon). Из контекстных диаграмм можно получить объекты аппаратного обеспечения и внешние интерфейсы. На основе диаграмм потоков данных можно выделить абстрактные типы данных и наборы данных. Типы данных можно получить также и из информационной модели. Диаграммы состояний позволяют идентифицировать активные объекты и основанные на объектах управляющие структуры.

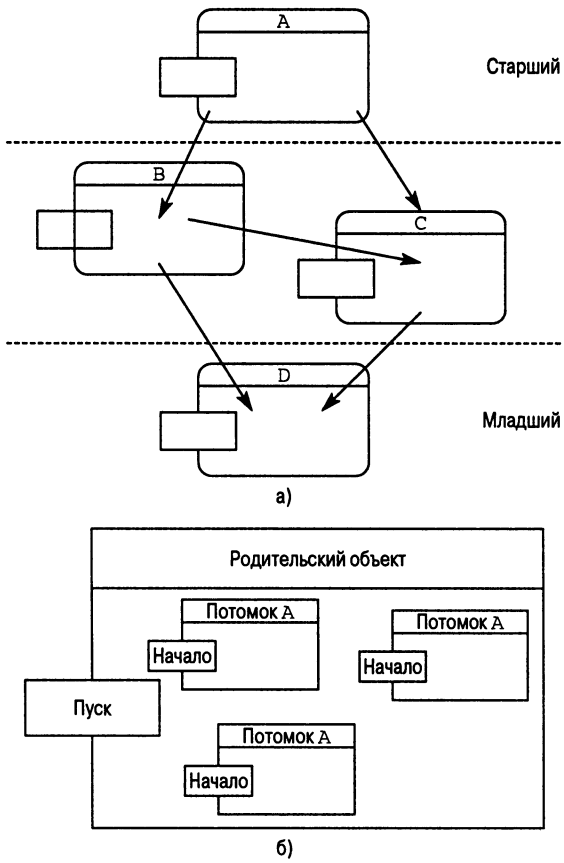


Рис. Б.5. В методе HOOD используются отношения "реализуется": а — иерархия старшинства с отношениями использования; б — родительский метод `Start` должен быть "реализован" методом `Begin` объекта `Child A`, который предоставляет всю требуемую функциональность метода `Start`

На втором шаге выбирается неформальная стратегия принятия решений. На этом этапе решается несколько задач: создается общий план на естественном языке, строятся приблизительные диаграммы метода HOOD и формируется описание текущего уровня абстракции.

На третьем этапе формализуется проектное решение. При этом идентифицируются и описываются объекты и их операции. Затем группируются объекты и создаются диаграммы метода HOOD, на которых отображаются отношения "родитель-потомок" и операции (композитная иерархия), отношения использования ("клиент/сервер" или старшинства), связи implemented by, потоки данных и исключительные ситуации. И наконец, любые проектные решения должны быть обоснованными. Для обеспечения возможности аудита принятых проектных решений рекомендуется также зафиксировать связи проектных решений с соответствующими разделами исходной спецификации требований.

Сейчас необходимо обратить внимание на терминологию. В HOOD методы называются "операциями", а связи "реализуется" указывают на то, что операции родительских объектов должны вызывать операции дочернего объекта.

После формализации стратегии принятия решения нужно формализовать само решение. Для достижения этой цели уточняется структура определения родительских объектов (ODS — Object Definition Skeleton), что предполагает определение типов, констант и данных. Кроме того, уточняется ODS-структура терминальных объектов, выполняется проверка на противоречивость, создается документация, а также генерируется и тестируется код на языке Ada.

Как уже отмечалось выше, отношения использования изображаются с помощью стрелок. Исключения обозначаются на стрелке черточкой, как показано на рис. Б.6, однако явно не отображаются. Другими словами, направление потока исключений на диаграмме не указано (неявно оно противоположно стрелке), и исключительная ситуация рассматривается в отношении объекта в целом, а не связывается с отдельным параметром или потоком данных. В следующем примере, если температура превышает некоторое предельное значение или возникает сбой в работе температурных датчиков, то можно сгенерировать исключение "перегрев" (overheated).

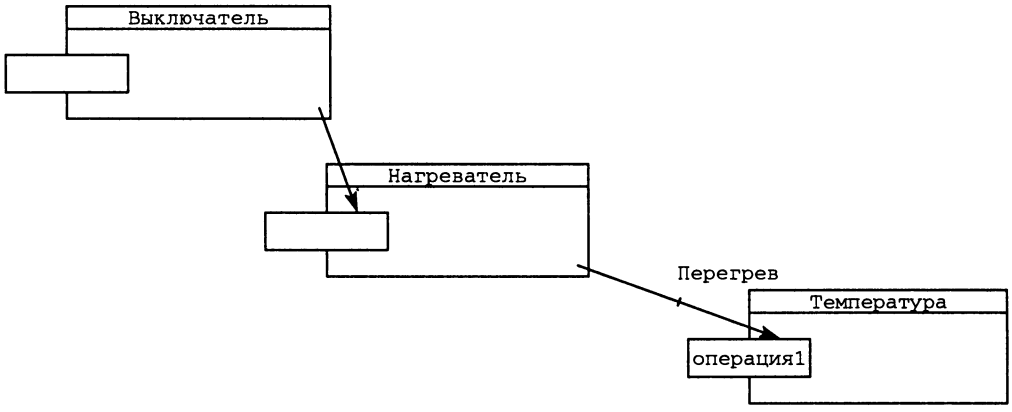


Рис. Б.6. Поток исключений, описанный в соответствии с методом HOOD. Исключительная ситуация `Overheated` генерируется объектом `Temperature` и обрабатывается объектом `Heater`

Кроме того, потоки данных могут изображаться с помощью условных обозначений, представленных на рис. Б.7, которые основаны на структурных диаграммах [819].

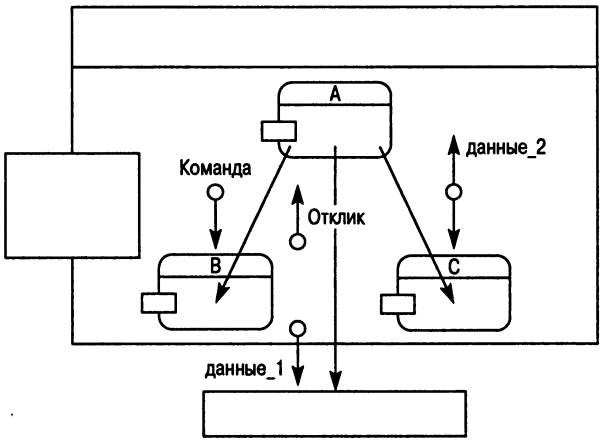


Рис. Б.7. Потоки данных в методе HOOD

В отношении метода HOOD в работе [389] были высказаны следующие критические замечания.

- Отсутствует поддержка универсальности, наследования и полиморфизма. Обычно программисты на языке Ada используют настраиваемые параметры, и это серьезный недостаток даже для метода, основанного на объектах.
- Описание объекта недостаточно отделено от его использования. Поэтому повторное использование поддерживается недостаточно.
- Система графических обозначений несовершенна и противоречива. Диаграммы в рамках метода HOOD явно не показывают, какие операции и какими объектами используются, какие потоки определены для каждой операции, как распространяются исключения и какие данные инкапсулируются объектами. В системе обозначений подпрограммы описываются как объекты, при этом управляющая структура также представляется как подпрограмма, которая передает управление задаче. Родительские объекты изображаются точно так же, как и объекты-потомки.
- Возникают трудности при построении иерархии "родитель-потомок". В системах реального времени события и исключения могут повлиять на глубоко вложенные подобъекты, поэтому передача управления по всем уровням иерархии оказывается неэффективной.

Помимо этого, многие специалисты считают, что метод HOOD, будучи пригодным для разработки многих приложений реального времени в военной области, гораздо меньше подходит для коммерческого использования. Недостаточная поддержка наследования делает его скорее основанным на объектах, чем объектно-ориентированным. Так, в рамках метода HOOD поддерживается повторное использование, однако не расширяемость. Кроме того, роль структуры данных (атрибутов сущностей) существенно нивелирована, поскольку основное внимание сконцентрировано на функциональных абстракциях. В следующем разделе будет рассмотрена система обозначений, которая отчасти позволяет решить некоторые из этих проблем.

OOSD

Некоторые из приведенных выше критических замечаний были учтены в другом методе проектирования — OOSD (Object-Oriented Structured Design) [776]. Строго говоря, OOSD — это не метод, а система обозначений, к которой можно добавлять методологические правила. Вероятно, среди рассмотренных выше методов эта система обозначений была ближе всего к объектному подходу, поскольку поддерживалось как наследование, так и абстракция. Хотя нотация OOSD гораздо полнее удовлетворяла принципам объектно-ориентированного подхода, на нее существенное влияние по-прежнему оказывал язык Ada. Подход OOSD предоставлял возможность постепенной адаптации для тех разработчиков, которые знакомы со структурным проектированием, поскольку именно на нем он отчасти и был основан.

Метод OOSD представлял собой систему обозначений, у которой не было конкретного владельца. Он был предназначен для архитектурного проектирования с использованием нисходящего структурного и объектно-ориентированного подходов. Метод OOSD разрабатывался для решения основных задач, присущих объектно-ориентированному проектированию: обеспечение повторного использования, модульности, расширяемости, а также представление наследования и абстракции. Этот метод должен обеспечивать визуальное представление интерфейсов между проектными компонентами, процесса генерации кода, независимость от конкретного языка программирования, а также способствовать взаимодействию разработчиков и пользователей и применению различных технологий. Весьма амбициозные цели!

В методе OOSD также используется нотация Буча, однако на него повлияли структурные диаграммы [819], а поддержка параллелизма основана на понятии мониторов [388]. Классы изображаются в виде прямоугольников, на которые накладываются более мелкие прямоугольники, обозначающие их методы. В литературе по OOSD атрибуты явно не упоминаются, однако для их обозначения можно использовать закрытую область хранения данных. Атрибуты неявно присутствуют также в потоках данных, которые изображаются в виде стрелок с незакрашенными кружками. Как показано на рис. Б.8, эти потоки обозначаются аналогично потокам на структурных диаграммах. Как и при использовании других методов в OOSD, круглые скобки обозначают перенаправление или передачу параметров.

Отношения "клиент/сервер" изображаются с использованием более жирных стрелок между объектами. Если с одной из таких стрелок связан выходной поток данных (как на рис. Б.9), то это означает, что объект-клиент (`save state`) инстанцирует объект.

В отличие от HOOD, в методе OOSD исключения объявляются явно. При этом для их обозначения используются ромбовидные области, наложенные на тело объекта. Для представления передачи параметров, связанных с исключительной ситуацией, применяются те же обозначения, что и для описания потоков данных. Однако в данном случае стрелки содержат заштрихованные ромбы (рис. Б.10).

На рис. Б.11 показаны закрытые операции, где операция `is full` (заполнен) используется методом `push` (добавить), который доступен за пределами стека.

Родовые (параметризованные) классы изображаются аналогичным образом, однако для представления прямоугольного контура тела класса используется пунктирная линия. Отношение наследования между классами изображается пунктирной линией со стрелкой, причем допускается множественное наследование.

Параллельные или асинхронные процессы обслуживаются мониторами, которые представляются параллелограммами (рис. Б.12). Понятие монитора сходно с понятием класса, однако его данные совместно используются его различными методами.

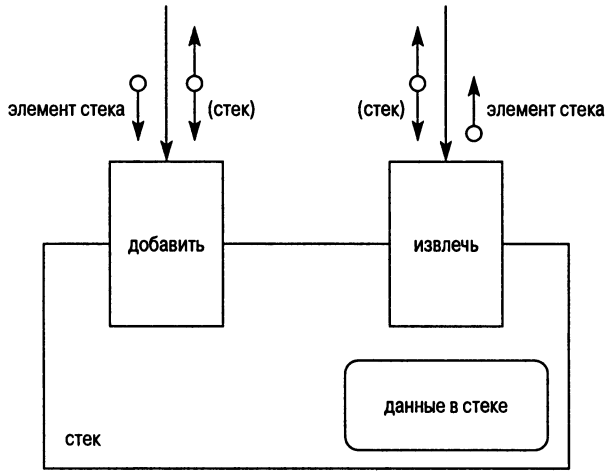


Рис. Б.8. Представление стека в виде объекта с использованием средств метода OOSD

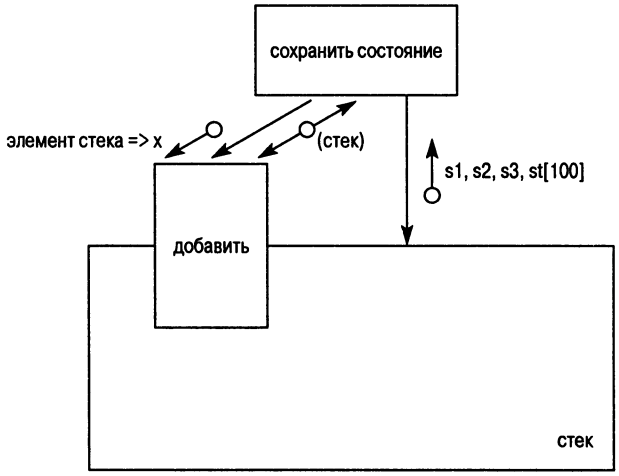


Рис. Б.9. Класс `save state` инстанцирует 103 объекта типа `stack` и может использовать их методы

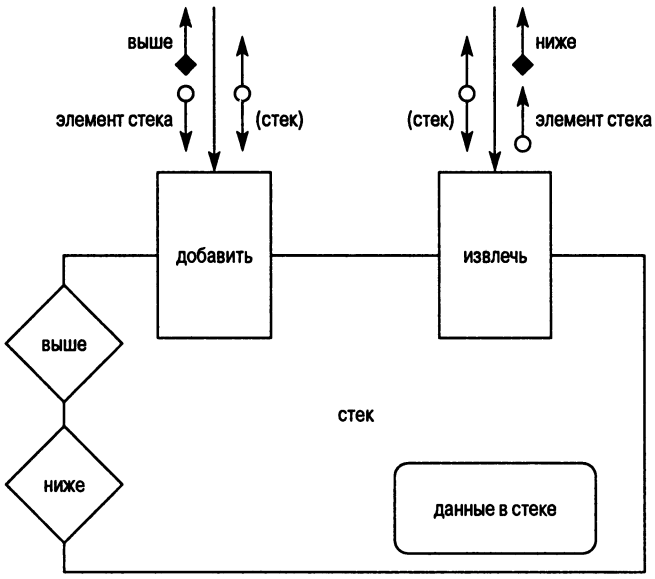


Рис. Б.10. Обработка исключений для объекта `stack`

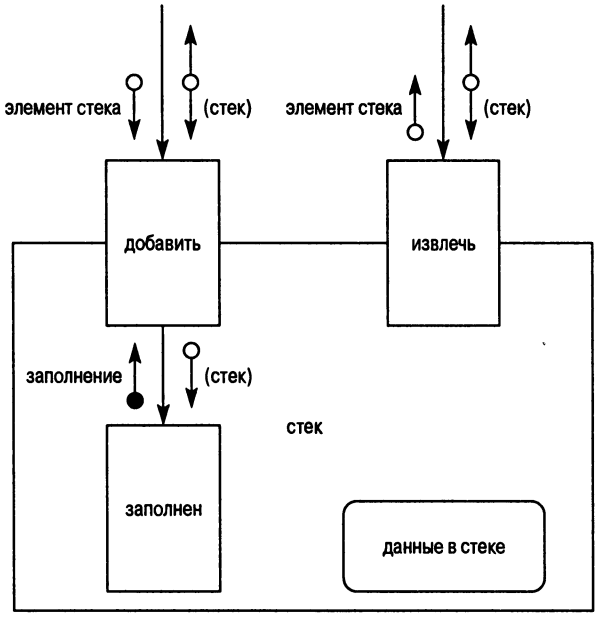


Рис. Б.11. Отображение скрытых операций в методе OOSD

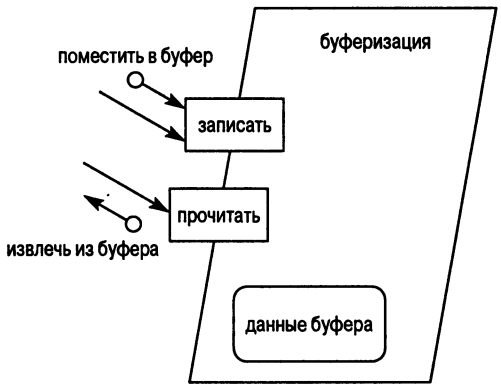


Рис. Б.12. Монитор с буферизацией

Полный набор символов OOSD представлен на рис. Б.13. Для получения более подробных сведений можно обратиться к работе [776], на которой и основан приведенный выше краткий обзор.

Вообще, OOSD не задумывался лишь как метод или система обозначений, предназначенная для поддержки методов объектно-ориентированного проектирования. При использовании OOSD можно добавлять новые правила проектирования. Там, где эти правила строго определены, их можно без особого труда формализовать, однако если правила достаточно расплывчаты (как некоторые требования метода HOOD) и нуждаются в принятии решения разработчиком, то можно прибегнуть к использованию экспертных систем.

Предполагалось, что метод OOSD не будет зависеть от языка Ada или какого-либо другого языка программирования. Это означает, что некоторые понятия метода OOSD могут явно не поддерживаться выбранным языком. Реализовать классы намного проще на языке C++, чем на языке FORTRAN, однако для этого подойдет любой из них. И опять же, на таких языках, как FORTRAN, которые не поддерживают параллельные процессы, гораздо сложнее реализовать мониторы. Точно так же с использованием метода OOSD нельзя представить некоторые детали языков. Например, в нем отсутствует эквивалент частично закрытых типов в языке Ada или виртуальных/дружественных функций в C++. Важно отметить также, что OOSD представляет собой систему обозначений скорее для архитектурного проектирования, а не для подробного физического проектирования. Это означает, что она не очень полезна при физической кластеризации файлов на дисках, выборе наиболее подходящего аппаратного обеспечения, обеспечении требуемого уровня обслуживания или оптимальном использовании памяти.

По поводу нотации OOSD важно отметить то, что ею готовы воспользоваться разработчики, уже знакомые со структурным проектированием, а благодаря поддержке концепции мониторов ее можно использовать для проектирования систем реального времени. Например, стрелки на диаграмме OOSD интерпретируются точно так же, как и на структурной диаграмме. Кроме того, в рамках OOSD определение объекта отделяется от его последующего использования другими объектами. Это означает, что система обозначений не требует указания всех деталей объекта при каждом его использовании. Немаловажное значение имеют также правила замещения методов при множественном наследовании. Если метод может быть унаследован двумя способами, то ему должно быть присвоено имя, облегчающее выбор более

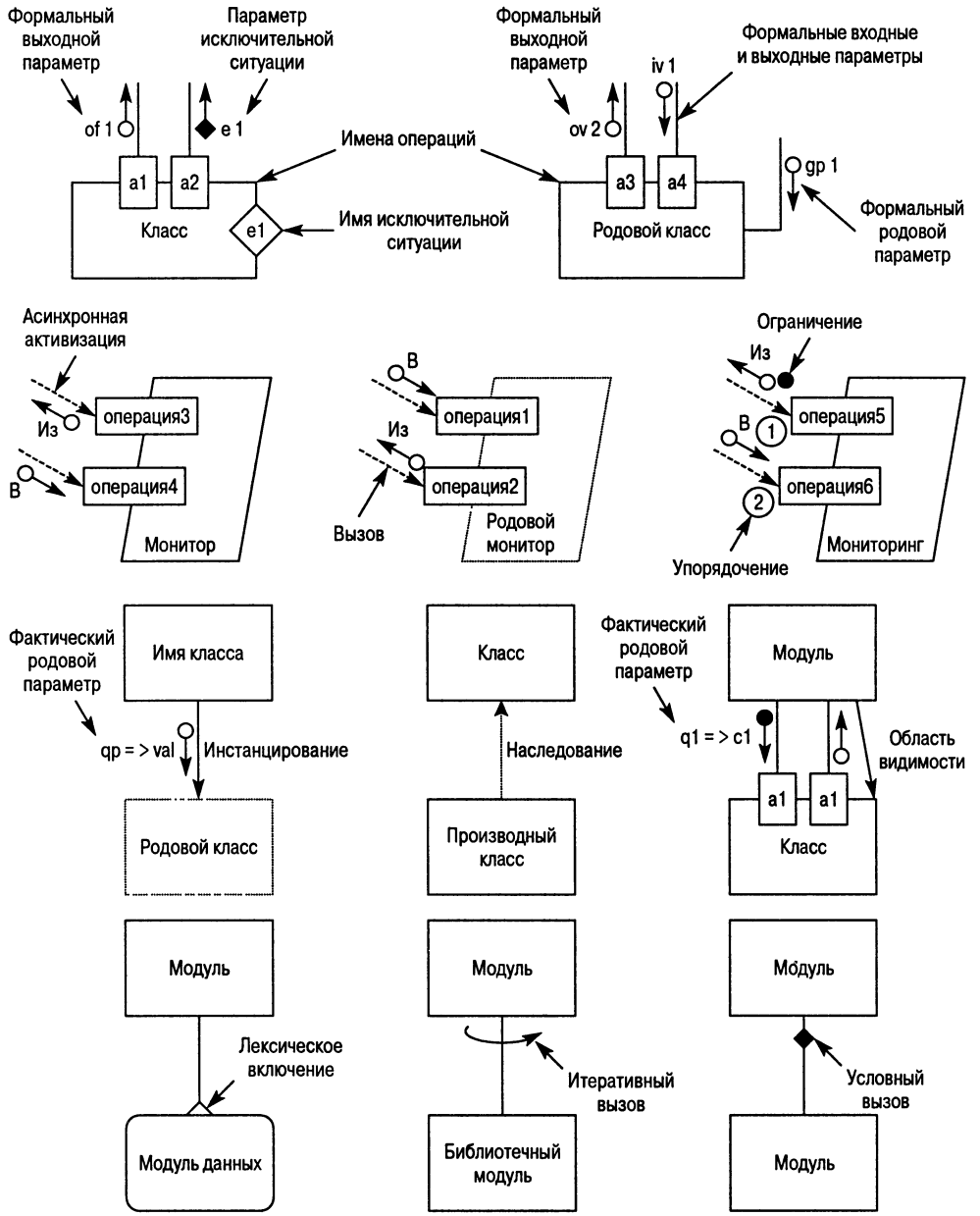


Рис. Б.13. Полная система обозначений OOSD

предпочтительного способа. Рассмотрим пример из предметной области геометрии. Класс геометрических объектов может иметь подклассы "симметричная фигура" и "твердое тело". Симметричное твердое тело описывается двумя этими классами. Теперь предположим, что

все геометрические объекты имеют метод "инвариантное вращение", однако для обычных твердых тел он намного проще, чем для симметричных фигур. Таким образом, симметричные твердые тела должны наследовать метод, обозначенный как "инвариантное вращение симметричных тел", или заместить его собственным методом в соответствии с требованиями решаемой задачи. В рамках подхода OOSD методы могут добавляться в подклассы или замещаться, однако они не должны удаляться.

OOSD был одним из наиболее передовых гибридных методов первого поколения, предназначенных для низкоуровневого объектно-ориентированного проектирования.

JSD и KISS

Метод разработки систем JSD (Jackson System Development) [410] следует отнести скорее к основанным на объектах, чем к объектно-ориентированным. JSD-модели строятся в терминах событий или действий и связанных с ними временны*х* зависимостей. В рамках этих событий метод JSD вначале позволяет выявить объекты. На следующем этапе в терминах взаимодействующих последовательных процессов, имеющих доступ к состояниям друг друга, разрабатывается спецификация. Таким образом, нарушается принцип сокрытия информации.

Между JSD и объектно-ориентированными методами проектирования существует много общего. В методе JSD определены полезные приемы, с использованием которых на этапе моделирования можно идентифицировать сущности и методы. Кроме того, методику анализа временно*г*о упорядочения можно рассматривать как средство документирования классов. Концепция объектов в методе JSD и объектно-ориентированном проектировании используется аналогичным образом, хотя в каждом из них применяется различная терминология. Метод JSD позволяет определить, какие объекты и операции являются существенными для решаемой задачи, особенно в тех случаях, когда очень важно учитывать временно*е* упорядочение событий. В остальных случаях подобный анализ несколько усложняется.

Джексон (Jackson) объявил о своей особой заинтересованности в решении проблемы, которая рассматривалась в главе 1. Она состоит в том, что наследование нарушает инкапсуляцию и, следовательно, не позволяет в полной мере воспользоваться повторным использованием. Вне всякого сомнения, идеи Джексона оказали и продолжают оказывать огромное влияние на разработку других объектно-ориентированных методов.

Несколько позже появились объектно-ориентированные версии метода JSD. Кроме того, предпринимались попытки создания объектно-ориентированных расширений обычных методов, в которых использовались приемы ELH (Entity Life History) и стиль JSD. Одним из таких методов является SSADM-4. Например, в этом методе EF-диаграммы (Effect Correspondence) применяются в качестве средства отображения видимости. Еще более экстремальные предложения сводились к тому, что наследование — это не более чем отношение "один ко многим". Следовательно, его можно рассматривать в рамках моделирования данных с использованием метода SSADM. Объектно-ориентированная версия метода SSADM была описана в работе [665].

В рамках метода объектно-ориентированного анализа и проектирования KISS [460] используется текстовый анализ и другие идеи, которые были позаимствованы из метода JSD, особенно его "истории жизни сущностей" (Entity Life History — ELH), в основу которых положена концепция регулярных грамматик. Эти ELH-истории создавались с использованием своеобразного набора "домино", являющегося альтернативой традиционным средствам отображения с помощью диаграмм. На метод KISS сильное влияние оказал схожий с SSADM

метод MERODE, в рамках которого историям жизни сущностей также уделяется особое внимание.

Все рассмотренные методы оказались крайне слабыми в области семантического моделирования данных. Ни один из подходов построения диаграмм (за исключением KISS) не предусматривал подробного описания атрибутов.

BOOCH91 и BOOCH93

В пересмотренном методе проектирования Буча [98] определено четыре основные модели и шесть представлений. На первых этапах рассматриваются статические аспекты системы — как логические, так и физические. Динамика исследуется с использованием существующих способов описания переходов между состояниями и построения временны*х* диаграмм. Схематически это можно представить в следующем виде.

Логическая структура	**Физическая структура**
Диаграммы классов	Диаграммы модулей
Диаграммы объектов	Диаграммы процессов
Динамика классов	**Динамика экземпляров**
Диаграммы состояний и переходов	Временны*е* диаграммы

Буч полагал, что как лингвистическая интерпретация в стиле Эббота (Abbott), так и традиционные приемы структурного анализа и объектно-ориентированный анализ были в равной степени хорошими предшественниками объектно-ориентированного проектирования. Однако одновременно с этим он предупреждал разработчиков, которые использовали структурный анализ, о том, что нужно сопротивляться "желанию вернуться к структурному анализу". Хороший совет на сегодняшний день.

Как классы, так и экземпляры изображаются в виде аморфных "облаков" (см. рис. Б.1), однако для представления классов используется пунктирная линия. Если эти "облака" имеют тень (как на рис. Б.1), то это свидетельствует о том, что они обозначают свободные подпрограммы, которые можно использовать в некоторых языках программирования.

Для изображения отношений между классами в нотацию Буча добавлена чрезвычайно богатая система обозначений. Эти условные обозначения представлены на рис. Б.14. Буч рекомендовал использовать несколько уровней, на каждом из которых содержалось бы несколько связанных между собой классов, образующих определенную категорию. Эти уровни изображаются в виде прямоугольников. Стрелки между такими прямоугольниками определяют видимость или отношения использования.

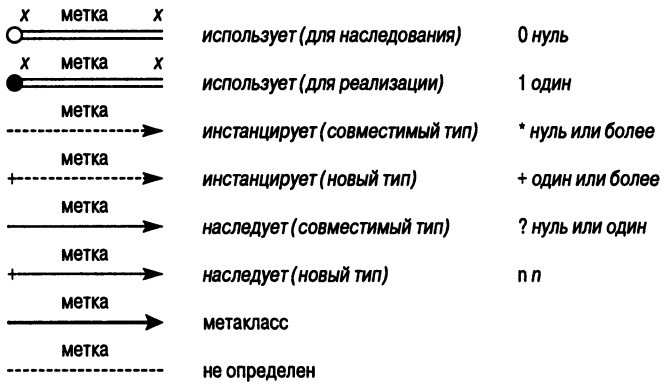


Рис. Б.14. Система обозначений для диаграмм классов в методе Booch91

В рамках метода Буча каждый класс описывается путем заполнения стандартного шаблона, в котором можно указать атрибуты, методы, а также следующую дополнительную информацию.

Описание (текст)
Видимость (экспортируется/импортируется)
Кратность (0/1/n)
Суперклассы
Используемые классы
Настраиваемые параметры
Реализация интерфейса (открытый/защищенный/закрытый)
Диаграмма состояний и переходов
Параллельность (последовательный/блокируемый/активный)
Постоянность (статический, динамический)
Пространственная сложность (текст)

Следует заметить, что выше представлена некоторая семантика данных, но не процессов, хотя эта информация может содержаться на диаграмме состояний и переходов, связанной с каждым классом. Некоторые разделы шаблона, например параллельность и пространственная сложность, связаны с достаточно низкоуровневым физическим проектированием.

На диаграммах классов и объектов описываются логический и статический аспекты программной системы. Физическое представление может отличаться от логического. Например, для обеспечения эффективного доступа можно реализовать кластеризацию данных. Для выполнения такого разделения (которое не упоминалось в более ранних работах) Буч отделил классы от модулей. Модули соответствуют программным сегментам, которые могут представлять собой отдельно компилируемые функции на языке C++ или сложные пакеты языка Ada. Условные обозначения модулей базируются на более ранней нотации, представленной на рис. Б.2, *б*. Диаграммы процессов, которые, по существу, представляют собой простые блок-схемы, описывают взаимодействие между физическими устройствами и процессорами.

С использованием метода Буча программную систему можно представить также в динамическом ракурсе. Это можно осуществить двумя способами. Диаграммы состояний и переходов

показывают динамику классов. Этот подход используется во многих методах объектно-ориентированного анализа, которые рассматриваются в данном приложении. Динамика на уровне экземпляров описывается при помощи временных диаграмм, позаимствованных из области проектирования аппаратных средств. Как показано на рис. Б.15, на временных диаграммах представляются методы каждого экземпляра, а также начало и конец их работы по отношению к методам других экземпляров.

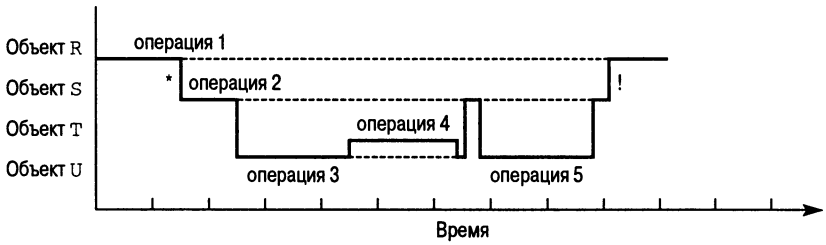


Рис. Б.15. Временная диаграмма, используемая в методе Booch91. Звездочка иллюстрирует создание экземпляра, а восклицательный знак — его разрушение

Временные диаграммы легки в управлении только для небольших систем, поскольку они сильно зависят от сложности взаимодействия процессов. В методе Objectory для представления динамики экземпляров используется аналогичный подход. В методе DEMOS [79] для этих же целей применяются диаграммы видов деятельности.

В то время метод Буча был одним из наиболее работоспособных методов проектирования и превосходил методы GOOD и HOOD, поскольку он не был связан с языком Ada и использовал намного более общее понятие структуры. Такие богатые семантические свойства метода позволяли с успехом его применять как аналитикам, так и проектировщикам. Это объясняется также инкрементным подходом к разработке программного обеспечения, приверженцем которого являлся Буч. В процессе прототипирования анализ и проектирование зачастую очень тесно переплетены. Основной недостаток метода Буча (что характерно и для других объектно-ориентированных методов проектирования) состоит в том, что глобальная динамика учитывалась в последнюю очередь. Более того, в рамках метода не предпринималось попыток серьезного рассмотрения бизнес-правил и других семантических аспектов спецификации. Помимо того, что Booch93 являлся наилучшим методом проектирования для программистов на языке C++ (и конечно, на языке Ada), он, на мой взгляд, несколько превосходил в этом отношении даже многие более поздние методы, в частности основанные на языке UML.

Одна из основных проблем проектирования программного обеспечения состоит в необходимости представления всей сложности принимаемых проектных решений. Объектно-ориентированные методы призваны решить именно эту задачу. Каскадные модели и другие бюрократические (т.е. "жесткие") подходы к построению программных систем не учитывают тот факт, что управление сложностью требует большой гибкости. Итеративные и управляемые рисками подходы основаны на осознании необходимости использования не только надежных, но и адаптируемых методов. Буч рекомендовал применять подход, который он называл "круговым проектированием". Другими словами, нужно реализовать процесс, состоящий из итеративных завершенных циклов, соответствующих разным уровням абстракции и

уточнения, который не выполнялся бы ни строго сверху вниз, ни строго снизу вверх. Ориентировочно этот процесс можно описать следующим образом.

- Идентифицировать классы и экземпляры.
- Определить их семантику.
- Выявить отношения между ними.
- Спроектировать программную систему в виде прототипа.
- Проверить связность и непротиворечивость системы.
- Переопределить классы, экземпляры, семантику и архитектуру программной системы на основе полученной информации.

Процесс прекращается после определения всех ключевых абстракций и функций. Определение методов объекта и его интерфейса может повлиять на другой объект или структуру классификации. Точно так же в процессе изучения структуры может оказаться, что необходимы новые объекты. И наконец, следует отметить, что лучшего или оптимального проекта не существует. Разные проектировщики могут разрабатывать различные модели, которые в равной степени будут соответствовать целям, преследуемым в процессе проектирования.

В методе Буча основной акцент делается на том, что идентификация объектов и соответствующих классов позволяет получить логическое проектное решение, которое не зависит от физического проектного решения, определяющего размещение этих классов в модулях.

В этом методе отдельно выделяются отношения "клиент/сервер" (т.е. старшинства или использования), а также отношения "родитель-потомок" (включения). На уровне классов отношение включения можно трактовать двояко: как классификацию (содержится понятие) и как композицию (содержится объект). Хотя в методе проектирования Буча и учитываются эти различия, однако ничего не говорится о взаимосвязи этих отношений. Отношения использования определяют управляющую структуру системы или топологию передачи сообщений.

Для объектов Буч выделяет три роли. **Исполнители** (actor), или активные объекты, могут воздействовать на другие объекты, но сами никогда не подвергаются воздействию со стороны других объектов. **Серверами** могут управлять только другие объекты. **Агенты** могут играть обе роли. Отношение "клиент/сервер" лучше всего рассматривать как относительное понятие. Объект может быть сервером в одном случае и исполнителем — в другом. Такой подход не имеет никакого отношения к системам на основе исполнителей, которые рассматривались в главе 3.

Включение объекта (например, в качестве значения атрибута) позволяет сократить количество объектов, видимых внешними объектами. Однако использовать такие объекты в других частях системы гораздо сложнее. При этом возможности повторного использования также оказываются весьма ограниченными. Таким образом, проектное решение играет чрезвычайно важную роль. При разбиении на объекты очень важно выбрать требуемую степень детализации. Как уже отмечалось, на этом этапе неопытные разработчики зачастую допускают ошибку, увлекаясь описанием слишком мелких деталей.

Метод Буча продолжал развиваться, и в 1993 году появилась его обновленная версия. По сравнению с предыдущими версиями, в новом методе акценты полностью сместились в сторону языка C++. В него были добавлены многие улучшенные условные обозначения. Некоторые из них отражали особенности языка C++, а другие были связаны с другими

методологиями. Однако сам метод по сути остался тем же. Метод "исчез со сцены" после того, как Буч стал одним из первых двух авторов языка моделирования UML.

Метод OODLE и рекурсивное проектирование

Подход к объектно-ориентированному анализу Шлеер (Shlaer) и Меллора (Mellor) будет описан в разделе Б.2. В этом же разделе рассматриваются два объектно-ориентированных компонента этого подхода. Система обозначений Шлеер-Меллора, как, впрочем, и большинства других методов, была вытеснена языком UML, однако основные методологические принципы все же не утратили своей оригинальности.

Язык OODLE (Object-Oriented Design LanguagE — язык объектно-ориентированного проектирования) является составной частью метода Шлеер-Меллора и предназначен для проектирования. В рамках этого подхода выделяется четыре типа диаграмм, взаимосвязь которых представляется на иерархической диаграмме, что облегчает создание документации и автоматическую поддержку. Эта система обозначений не зависит от конкретного языка программирования, хотя в ней можно заметить некоторое смещение к языку Ada, а поддержка дружественных отношений напоминает язык C++. Диаграммы могут быть следующих четырех типов.

- **Диаграммы зависимостей** иллюстрируют отношения использования (клиент/сервер) и дружественные отношения между классами.
- **Диаграммы классов** показывают внешнее представление класса, аналогичное предложенному в [129].
- **Структурные диаграммы классов** отображают структуру методов класса, а также потоки данных и управления.
- **Диаграммы наследования** представляют иерархию наследования.

На рис. Б.16 схематично представлены все типы диаграмм. На диаграмме зависимостей одинарные стрелки обозначают сообщения, а двойные стрелки указывают на нарушение инкапсуляции дружественными функциями. На диаграмме классов в верхнем прямоугольнике отображается имя класса, а во внутренних — его методы. Шестиугольники, присоединенные к обозначению метода с внешней стороны, иллюстрируют передачу параметров. Если они находятся внутри обозначения класса, то описывают скрытые структуры данных. Прямоугольники на структурных диаграммах классов обозначают методы, а шестиугольники — передачу параметров между ними (подобно стилю, принятому в рамках метода OOSD). Язык OODLE предоставляет гораздо более обширную систему обозначений, чем было упомянуто выше. Он позволяет отразить полиморфизм, исключения и т.д.

Некоторые проблемы проектирования привели к возникновению концепции рекурсивного проектирования. Зачастую классы не очень хорошо согласуются друг с другом. Кроме того, несмотря на то что повторное использование основано исключительно на внешних интерфейсах, разные стили реализации различных объектов приводят к возникновению трудностей при сопровождении. Поэтому проектирование должно быть не итеративным, а рекурсивным. Кроме того, по возможности оно должно основываться на методе проектирования с четкой семантикой. Рекурсивное проектирование — это абстрактный принцип, а не сам процесс проектирования.

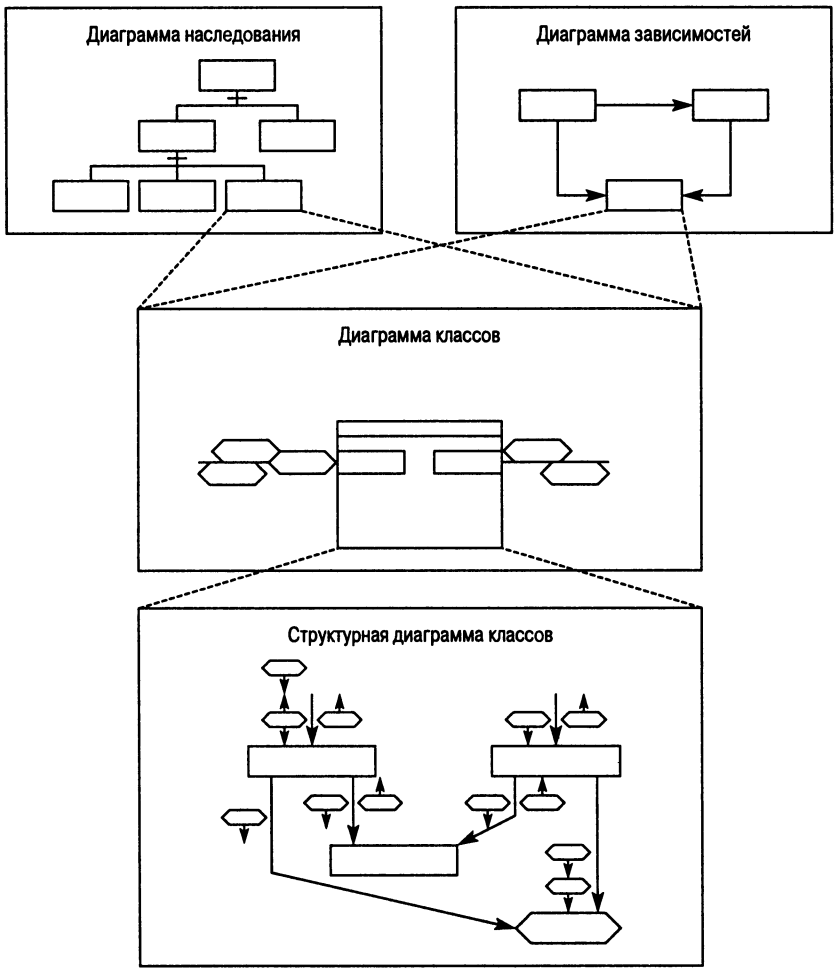


Рис. Б.16. Упрощенные диаграммы OODLE и их взаимосвязь

Основная (и неплохая) идея состоит в том, что вопросы производительности и удобства поддержки системы должны решаться путем применения набора общих правил ко всем модулям с исходным кодом, а не в результате настройки системы на низком уровне и на скорую руку. При использовании второго подхода каждый компонент превращается в особый случай. В этом и состоит основная мудрость, хотя в большинстве объектно-ориентированных проектов применяются другие подходы, которые контрастируют с подходом Буча и методами, основанными на UML. Однако на это можно возразить следующее: при повторном использовании новые проекты сами являются особыми случаями. Рекурсивный процесс разработки требует, чтобы приложение было отделено от различных предметных областей, в которых оно повторно используется, и чтобы на любом уровне создавались формальные способы обмена данными. Например, можно сконструировать стандартный метод передачи данных между несколькими пользовательскими интерфейсами и определенной операционной системой. Ключевым

является понятие "архитектуры", которое позволяет отделить приложение от конкретных реализаций (рис. Б.17). Любопытно, что именно этого и пытались достичь проектировщики NeXtStep и Taligent на уровне операционной системы. Метод HOOD позволяет создать подобный "мост", однако лишь для обеспечения многозадачности в языке Ada. А это нельзя отнести к достаточно общему случаю.

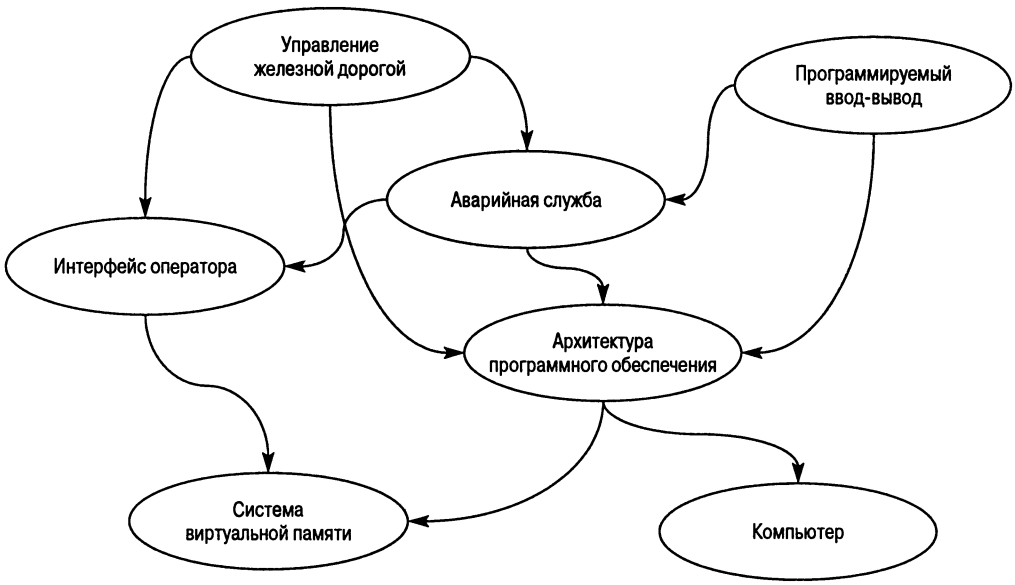


Рис. Б.17. Типичная диаграмма предметной области Шлеер-Меллора

И наконец, в табл. Б.1 представлено сравнение рекурсивного и итеративного процессов разработки.

На основе табл. Б.1 можно сделать вывод о том, что длительность всех этапов итеративного процесса разработки пропорциональна сложности требований, тогда как в процессе рекурсивной разработки затраты на создание кода и проектирование остаются постоянными. Причем после начальных более высоких капиталовложений преимущества повторного использования становятся все более ощутимыми по мере увеличения количества успешных проектов.

В данном случае термин "генерация" скорее означает кодирование на основе правил "вручную". Однако для достижения поставленных целей наиболее естественно воспользоваться соответствующими CASE-средствами. По мнению автора, принципы рекурсивного проектирования можно без труда применять и в рамках традиционного подхода к разработке программных систем.

Таблица Б.1. Сравнение итеративного и рекурсивного процессов разработки

Итеративный процесс	Рекурсивный процесс
ДЛЯ КАЖДОГО требования ВЫПОЛНИТЬ анализ проектирование кодирование выполнение кода (тестирование)	ДЛЯ КАЖДОГО требования ВЫПОЛНИТЬ анализ построение модели анализа (тестирование)
END DO	END DO
выполнить интеграцию программного обеспечения	определить правила проектирования протестировать правила применить правила для "генерации" кода

Карты CRC и методология RDD

Подход к анализу и проектированию на основе обязанностей с применением карт CRC (Class, Responsibility and Collaboration) был предложен в [64]. Этот подход достаточно полно описан в работе [803]. Его удобно использовать для документирования объектно-ориентированных проектных решений, а также для обучения базовым понятиям. Эта методология известна также как RDD (Responsibility Driven Design — проектирование на основе обязанностей). Она обладает единственным преимуществом, которое заключается в применении для моделирования учетных карточек, хотя первоначально планировалось использовать гипертекстовую систему. На практике набор карточек оказался более эффективным.

В рамках подхода на основе CRC-карточек предполагается, что существует спецификация требований, а для выявления ключевых объектов выполняется текстовый анализ, напоминающий анализ Эбботта. Для каждого объекта, представляющего некоторый класс, готовится карточка, на которой содержится имя класса, а также перечень его членов и суперклассов. В ходе последующего текстового анализа формулируются "обязанности" или перечень методов, необходимых данному классу. В дальнейшем эти методы уточняются. При этом рассматриваются структуры трех видов: классификации, композиции и (с этим мы сталкиваемся впервые) аналогии. Отношение использования определяется в виде списка классов, с которыми "сотрудничает" каждый класс. Эта информация также размещается на карточке. После этого идентифицируются отношения использования, а также отношения "обладает знаниями о" и "зависит от". Анализ "сотрудничества" позволяет управлять степенью детализации, поскольку классы, которые не сотрудничают с другими классами, на этом этапе отбрасываются. Далее анализ связан с установлением различий между абстрактными и конкретными классами, а для описания совместно используемых методов применяется теоретико-множественная система обозначений. Классы упорядочиваются по слоям (или подсистемам), которым назначаются контрактные отношения с другими подсистемами и списки делегируемых задач. И наконец, все компоненты, классы, методы, подсистемы и контракты проектируются со всеми деталями.

В рамках этого метода особое внимание уделяется обязанностям (т.е. методам), а не атрибутам (т.е. данным), что позволяет отодвинуть необходимость принятия как можно большего количества решений, связанных с реализацией. Однако это вовсе не означает, что отрицается существование атрибутов. Этот метод может применяться как на уровне анализа, так и на

уровне проектирования. Причем используемые в технологии RDD принципы можно применять в качестве набора правил в рамках других объектно-ориентированных методов анализа. При автономном использовании подход CRC лишь в незначительной степени способствует идентификации объектов и описанию управляющих структур. Контракты можно применять для определения пред- и постусловий, однако поддержка бизнес-правил и функциональной семантики весьма незначительна.

Формально обязанности представляются в виде атрибутов и подразделяются на два типа: **обязанности знания**, т.е. состояние объекта, и **обязанности действия**, т.е. операции, которые объект может выполнять. **Взаимодействие** предусматривает передачу запросов к серверу для получения помощи в выполнении клиентом своих обязанностей. На более детализированном уровне оно основывается на видимости или отношениях использования. С использованием CRC-карт ассоциации обрабатывать не очень удобно. **Соглашение**, или **контракт**, представляет собой набор сообщений, которые клиент может передавать серверу, и основывается на отношениях использования высокого уровня. В некоторых случаях контракты полезно применять для группировки взаимосвязанных обязанностей.

На рис. Б.18 представлена типичная CRC-карточка и карточка для подсистемы. Формирование списка членов или подклассов — это не очень хорошая идея, поскольку при этом нарушается инкапсуляция и, следовательно, повторное использование. Графически взаимодействие можно представить в виде диаграммы, как показано на рис. Б.19. В рамках описываемого метода такие диаграммы применяются для выделения связанных подсистем и групп, соответствующих контрактам. Практически для любой системы подобные диаграммы становятся очень громоздкими. Так что несмотря на то, что сама идея достаточно хороша, применить ее на практике непросто. Для группировки и разбиения на уровни необходимо воспользоваться другими методами.

При использовании CRC-карт в качестве входных данных используются письменно сформулированные требования, после чего выполняются следующие шаги.

1. Идентифицировать объекты (на основе существительных, содержащихся в спецификации) и сгруппировать их в виде иерархии.
2. Выявить обязанности (на основе глаголов, содержащихся в спецификации).
3. Назначить классам обязанности.
4. Исследовать построенную классификацию для уточнения обязанностей.
5. Выявить взаимодействие между классами.
6. Отбросить классы, не участвующие во взаимодействии.
7. Уточнить построенную иерархию классов.
8. Сгруппировать обязанности с использованием контрактов.
9. Определить подсистемы на основе анализа взаимодействия.
10. Выполнить более детальное проектирование.

Многие пользователи метода RDD выполняют "сквозной" контроль, чтобы проверить построенную модель и исследовать глобальную динамику системы. Лишь несколько методов позволяют достаточно хорошо решить эту задачу.

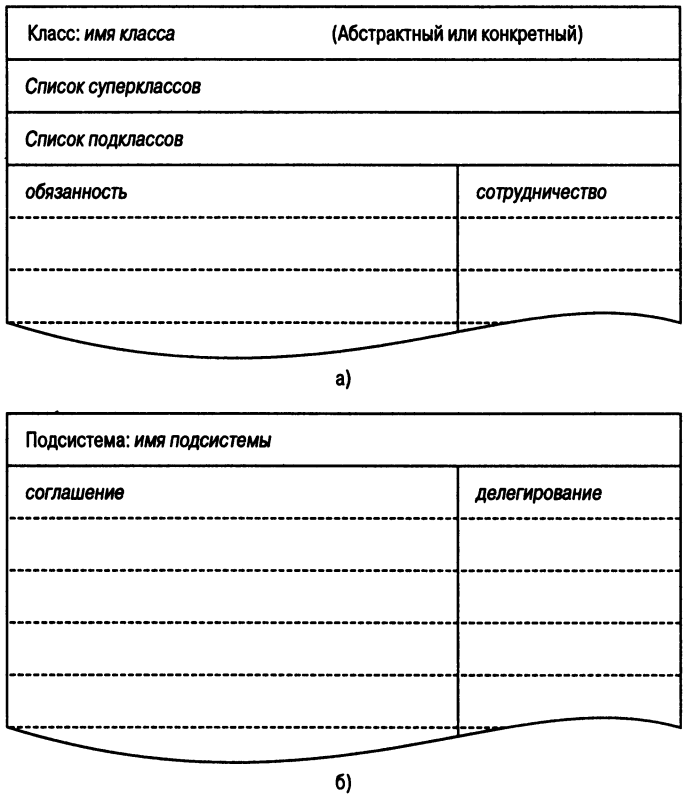


Рис. Б.18. CRC-карты: а — для класса; б — для подсистемы

RDD представляет собой упрощенный, но весьма практичный метод. Многие из его условных обозначений используются в качестве основы для других средств представления. Этот подход оказывается прекрасным инструментом для обучения искусству проектирования и позволяет быстро получить необходимые навыки. Усовершенствование методики использования CRC-карт в контексте формулировки требований и анализа более подробно рассматривалось в главе 7.

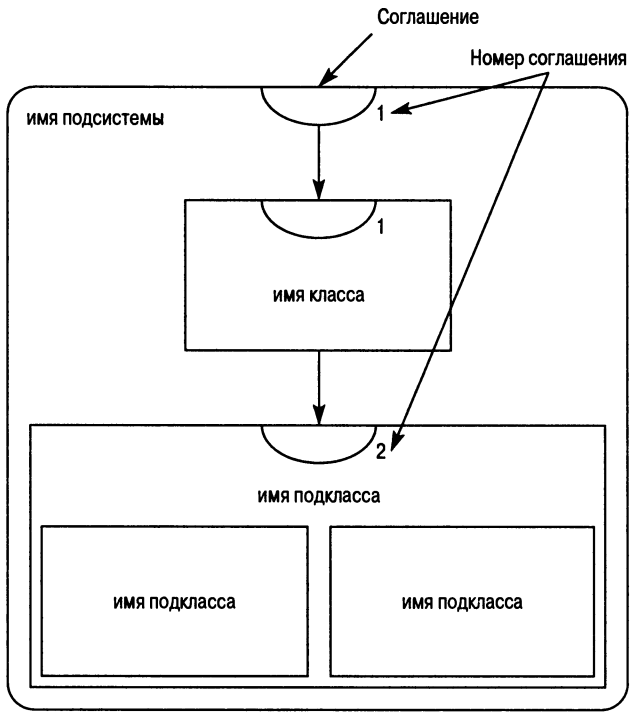


Рис. Б.19. Пример диаграммы взаимодействия

Б.2. Ранние методы объектно-ориентированного анализа

Методы анализа появились позже методов объектно-ориентированного проектирования, поэтому в своем описании я придерживался хронологического порядка.

Метод OOSA Шлеер-Меллора

Рассматриваемые ниже методы анализа появились позже методов объектно-ориентированного проектирования. Их обзор в данном разделе приводится в хронологическом порядке.

Самая первая работа, содержащая в своем названии слова "объектно-ориентированный анализ", появилась у Салли Шлеер и Стива Меллора [707]. Однако в реальности этот метод не мог считаться объектно-ориентированным по нескольким причинам, в том числе из-за полного отсутствия поддержки концепции наследования. Описываемый в этой публикации метод можно рассматривать как нечто большее, чем простое расширение моделирования данных на основе объектов. Однако позже [708] было введено понятие наследования (выделение подтипов элементов) и описан подход, согласно которому методы можно обнаруживать в процессе моделирования жизненных циклов сущностей с использованием диаграмм переходов. Более того, в рамках метода Шлеер и Меллора к объектам можно применять

правила нормирования (правила назначения атрибутов), что позволяет рассматривать их как нечто большее, чем простые реляционные таблицы.

Первый этап метода Шлеер-Меллора заключается в определении объектов и их атрибутов. При этом система обозначений моделирования сущностей основывается на представлении Уорда/Меллора (Ward/Mellor). Затем основное внимание уделяется описанию жизни объектов, при этом используются диаграммы перехода из состояния в состояние в стиле Мура (Moore) и соответствующие таблицы. Далее они применяются для определения операций. Система обозначений модели состояний, по существу, ничем не отличается от других аналогичных методов. Управление глобальной динамикой основано на манипулировании жизненными циклами объектов, представленными в моделях объектного взаимодействия. Эти модели иллюстрируют события, возникающие при взаимодействии сущностей (или передачу сообщений между ними). Кроме того, на основе модели жизненных циклов можно выделить также и отношения между объектами.

В рамках метода OOSA создается информационная модель (или модель данных) с объектами, атрибутами и отношениями. Затем строится модель состояний, на которой отображаются состояния объектов и переходы из одного состояния в другое. И наконец, с использованием диаграммы потоков данных определяется модель процессов. На этот метод сильное влияние оказало реляционное проектирование. Объекты представляются в первой нормальной форме, а их тождественность не обеспечивается принимаемыми проектными решениями. Как правило, приверженцами метода OOSA являются те пользователи, которые раньше использовали подход Уорда/Меллора (Ward/Mellor). Так сложилось, что этот метод применяется при моделировании систем реального времени или при управлении технологическим процессом, а не при разработке коммерческих программных систем обработки данных. Метод OOSA является комплексным; в рамках этого метода выполняется и анализ, и проектирование. Он прошел сложный путь развития, но в свое время все же оказался достаточно популярным. Аспекты проектирования, характерные для этого метода, и соответствующая система обозначений уже рассматривались выше. Поэтому повторно обсуждать эти вопросы нет необходимости.

Оригинальная идея, заложенная в метод OOSA, заключается в определении блоков повторного использования (как было проиллюстрировано на рис. Б.17). В процессе проектирования программного обеспечения это позволяет использовать многоуровневый подход и все преимущества повторного использования. Этот метод поддерживается различными CASE-средствами и в настоящее время по-прежнему широко используется.

Метод Коада

Другой подход, во многом вобравший в себя традиции моделирования отношений "сущность-связь", был предложен Коадом и Йордоном [171–173]. Главным образом, этот метод интересен как первый простой, достаточно полный и практичный объектно-ориентированный метод анализа, предоставляющий систему обозначений, которую можно применять в рамках коммерческих проектов. По сравнению с представлением Буча, Шлеер-Меллора и многих других объектно-ориентированных методов, Коад и Йордон ввели более четкую систему обозначений. При этом виды деятельности в значительной степени сместились в сторону анализа, а не проектирования. Структуры данных языка Ada, модульная декомпозиция и другие языковые конструкции в данном методе не имеют никакого значения. Он чрезвычайно прост, хотя в некоторых областях являлся недостаточно совершенным.

Одним из наиболее примечательных свойств представлений Шлеер-Меллора и Коада-Йордона было явное задание атрибутов. Выше уже было показано, как при этом нарушается классический объектно-ориентированный принцип, в соответствии с которым объекты должны задаваться исключительно при помощи их методов. Однако атрибуты могут идентифицироваться стандартными методами, имеющими доступ к их состоянию, например "записать величину зарплаты" и "извлечь величину зарплаты". Сложность структур данных, которая характерна подавляющему большинству коммерческих систем, приводит к необходимости явного задания атрибутов.

Книга Коада и Йордона, посвященная вопросам анализа, дважды переиздавалась в течение года. Позднее Йордон отказался от этого метода и разработал другой [820]. Новый метод получил название метода "основных объектов" (или Mainstream Objects) и имел много общего с методом SOMA. В результате метод Коада-Йордона стал ассоциироваться с именем Коада.

Коад разделил анализ на пять этапов и обозначил их следующим образом.

- *Части*. В проблемной области выделяются "части", соответствующие "уровням" или "слоям" диаграмм потоков данных, а также подсистемам Буча или категориям классов. Эти части должны допускать простоту управления, поэтому в них может содержаться примерно от пяти до девяти вложенных объектов.

- *Объекты*. После выделения уровней идентифицируются объекты. Для этого выполняется поиск бизнес-сущностей. Этот процесс во многом выполняется точно так же, как и анализ данных. Для успешного решения этой задачи Коад и Йордон сформулировали дополнительные рекомендации.

- *Структуры*. Определяются две совершенно различные структуры классификации и композиции. В тех случаях, когда наследование связано с классификацией, такие структуры представляют собой деревья специализации/обобщения. Коад совсем немного сообщает о том, как эти деревья должны обрабатываться или связываться друг с другом.

- *Атрибуты*. Как и при стандартном анализе данных, подробно описываются атрибуты, а затем с использованием расширенного реляционного анализа (Extended Relational Analysis — ERA) определяются отношения модальности и множественности. В этом состоит основное отличие этого метода от некоторых других объектно-ориентированных подходов к проектированию, которые лишь определяют методы.

- *Службы*. Путем идентификации служб Коад предлагает идентифицировать операции. В объекте любого типа должны определяться методы, предназначенные для создания и удаления экземпляров, извлечения и записи значений, а также специальные методы, определяющие поведение этого объекта.

В работе [172] перечисленные выше пять видов деятельности (или, по словам Коада, слоев) были дополнены четырьмя новыми компонентами. В общем, подход Коада к объектно-ориентированному проектированию заключается в преобразовании результатов объектно-ориентированного анализа и получении так называемого компонента PDC (Problem Domain Component). Для этого в пространство решения добавляются новые классы, а также три новых компонента: взаимодействия с человеком (Human Interaction Component — HIC), управления задачей (Task Management Component — TMC) и управления данными (Data

Management Component — DMC). Эти компоненты позволяют добавлять на диаграммы объектно-ориентированного анализа специфичные для этапа проектирования характеристики, например потоки управления, специальные процессоры, программные пакеты и т.д. При этом большой акцент сделан на использовании в процессе анализа и проектирования схожей системы обозначений, а также на сглаживании перехода от анализа к проектированию, который в традиционных методах является несколько резким. Эта статья Коада, связанная с объектно-ориентированным проектированием, появилась в ответ на критические отзывы по поводу метода объектно-ориентированного анализа. Например, для слоя служб были введены ограничения на запуск и завершение. Однако логическое и физическое проектирование реально не были разграничены. В некоторых критических замечаниях метод объектно-ориентированного анализа Коада упрекается в неспособности обработки динамических свойств. Возможно, именно поэтому на странице 151 работы [172] можно найти следующую фразу: "... моделируется динамика поведения. Входящие и исходящие сообщения можно выделить, выбрав службу или службы. С другой стороны, можно показать потоки выполнения (по одному или все сразу), воспользовавшись для различных потоков разными линиями". Однако при этом не совсем очевидно, должны ли нумероваться отдельные экземпляры сообщений или каким образом можно проверить полноту проектного решения.

Как видно из рис. Б.20, *а*, для представления типа объекта Коад и Йордон предлагают использовать условное обозначение, состоящее из трех частей.

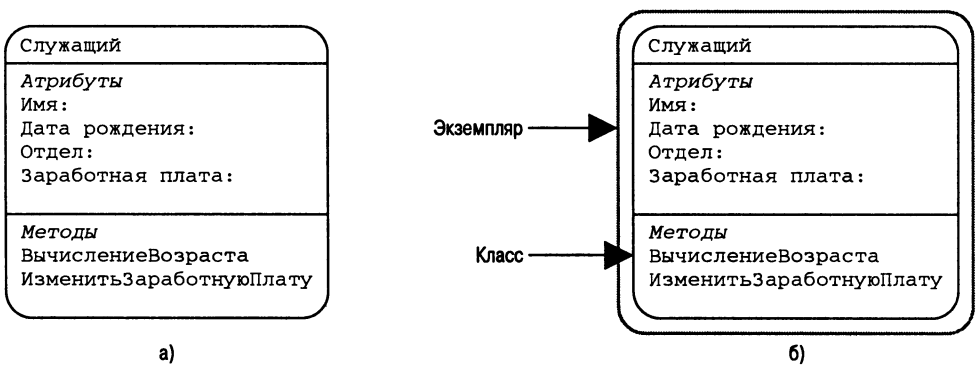


Рис. Б.20. Представление стандартного объекта в системе обозначений Коада-Йордона (а); обновленное представление требует наличия внешнего серого контура вокруг обозначений объектов, которые могут иметь экземпляры (б)

Вскоре в компании Коада, Object International, было разработано недорогое CASE-средство OOATool. Позднее его заменил программный продукт *Together*.

Передача сообщений между объектами изображается точно так же, как и в нотации Буча, — стрелками. В этой области подход Буча выглядит более очевидным, чем метод Коада. Автору кажется, что предпочтительнее применять именно его терминологию клиентов и серверов. Вне всякого сомнения, более поздняя система обозначений Буча (см. рис. Б.14) является наиболее удачной. Как и Буч, Коад тоже позволяет нумеровать сообщения. Этот подход удобен при первоначальном анализе, однако он приводит к неразберихе при последующем усложнении системы. Получающиеся в результате "спагетти" оказываются избыточными,

поскольку уже имена самих методов иллюстрируют, какие сообщения будут обрабатываться. Достаточно показать лишь видимость, т.е. *может* ли сообщение быть передано.

Во второй редакции своей книги Коад признал справедливость критических замечаний по поводу разграничения классов и экземпляров. Как показано на рис. Б.20, *б*, для обозначения типичного экземпляра класса он воспользовался серым контуром вокруг изображения класса, который может иметь хотя бы один экземпляр. При этом считается, что сообщения передаются экземплярам и соответствующие линии заканчиваются на внешнем прямоугольнике. Иерархия использования также оказалась достаточно запутанной, поскольку стрелки иллюстрируют передачу реальных сообщений, а не отношения "клиент/сервер" между классами.

Разбиение на части — это процесс, который выполняется на разных этапах анализа и может быть использован для первоначальной декомпозиции либо для построения модели после идентификации объектов или уточнения их перечня (рис. Б.21). Аналогичный процесс обычно связан и с построением диаграмм потоков данных. Части изображаются в виде прямоугольников, а передача сообщений иллюстрируется стрелками, как если бы слои были объектами. Однако в рассматриваемом методе части определяют произвольную декомпозицию, которая облегчает понимание, однако имеет нестрогую семантику. Еще хуже то, что сообщения или даже связи наследования способны пересекать границы объектов, хотя это и лишено здравого смысла.

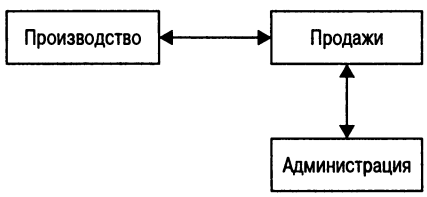


Рис. Б.21. В идеале, части должны представлять собой группы, содержащие от пяти до девяти объектов, или, возможно, согласованные структуры классификации или композиции

Для представления иерархии наследования Коад предложил использовать условные обозначения, представленные на рис. Б.22. Представление композиции показано на рис. Б.23.

Если рассматривать эти иерархии в качестве составных частей, обычно считается, что сообщения передаются к вершине иерархии композиции и к листьям иерархии классификации. Однако всегда будут существовать ситуации, которые такому подходу не будут удовлетворять. Сообщения и отношения показаны только для экземпляров. Проблема, появившаяся во втором издании книги Коада, заключается в существовании различий между тремя видами композиции: часть-целое, контейнер-содержимое и коллекция-члены. В этом состоит фундаментальная ошибка представления семантических примитивов. Последние два из них являются лишь дополнительными структурами, а не отдельными случаями композиции. Такие структуры вполне могут использоваться, но их не следует путать с иерархиями композиции. В конкретной предметной области действительно могут встретиться другие важные структуры. Например, для базы данных антропологических исследований в качестве основополагающей структуры можно с успехом использовать родство.

При определении атрибутов подобные соображения применимы к любым задачам моделирования данных. Аналитик может воспользоваться преимуществом наследования и в определенных случаях предположить, что неопределенные элементы являются унаследованными.

В версии метода 1990 года не поддерживается множественное наследование атрибутов (или методов). В обновленной версии множественное наследование уже можно использовать, однако при этом осталась нерешенной проблема разрешения конфликтов, когда для каждого одиночного атрибута или метода можно точно указать необходимые комментарии. Исходя из этого можно сделать вывод о том, что даже в исправленном методе множественное наследование не поддерживается должным образом.

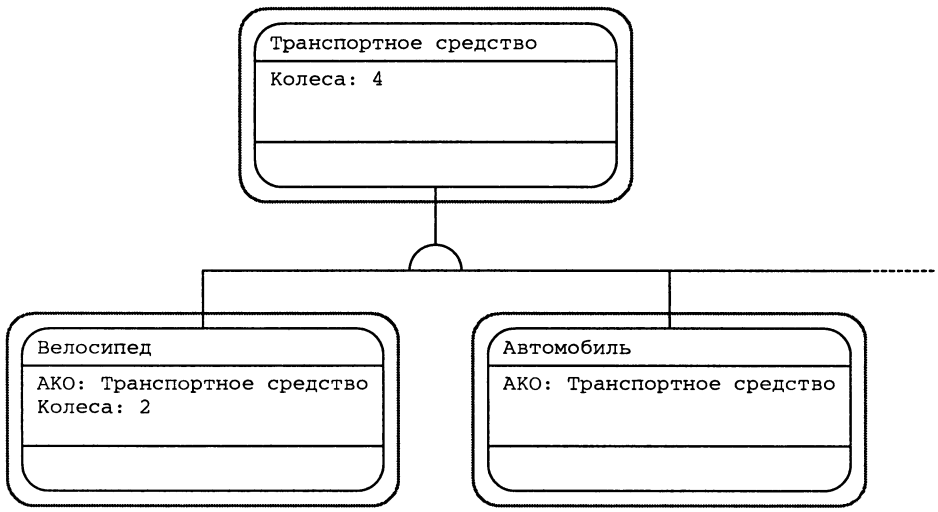


Рис. Б.22. Иерархия наследования в нотации Коада

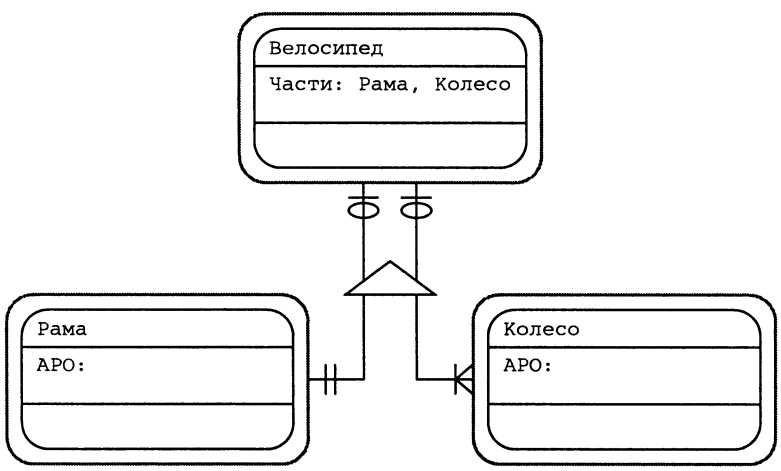


Рис. Б.23. Структура композиции в нотации Коада

Одно из самых полезных предложений Коада было связано с использованием стандартных объектов и шаблонов описания методов. В основу этого подхода положен язык формальных описаний Ina Jo [798]. Однако вместо него можно воспользоваться и любым другим стилем описания программных шаблонов, который позволит облегчить создание понятной документации и разработать полное описание как объектов, так и методов. В шаблоне должно определяться имя объекта, а также перечень имен атрибутов и методов точно так же, как и на диаграммах. Однако в шаблоне должны быть описаны и свойства атрибутов. Кроме того, в нем могут содержаться комментарии разработчика, а также информация о трассируемости, критических режимах, ограничениях и т.д. В шаблоне на английском языке или в виде структурированных утверждений задается цель каждого метода. Конечно, в случае необходимости текст может сопровождаться и диаграммами. По существу, каждый метод определяется точно так же, как аналитики всегда описывали компоненты системы. Единственное отличие состоит в том, что все эти детали будут скрыты внутри объекта и доступны только через его интерфейс.

Метод Коада оказывается несовершенным при описании динамики системы. Хотя для представления динамики объектов предлагается аппарат описания переходов из состояния в состояние, однако соответствующая система обозначений не определена. Кроме того, не обеспечивается также и явная поддержка описания бизнес-правил.

Подход Коада имеет и некоторые другие недостатки. Под влиянием развития реляционных баз данных Коад считал, что атрибуты должны быть "атомарными". Однако такой подход нельзя считать правильным. Объектно-ориентированные методы предназначены для моделирования сложных объектов, которые не должны быть атомарными. Кроме того, он отдает предпочтение использованию ключей, а не обеспечению идентичности объектов и нормализации.

На рис. Б.24 представлена большая часть нотации Коада-Йордона, описанной во втором издании их книги, посвященной вопросам анализа. На приведенном рисунке демонстрируются результаты анализа предметной области, связанной с регистрацией транспортных средств. На диаграмме содержится два слоя, одна иерархия композиции и несколько иерархий классификации. Толстые линии иллюстрируют передачу сообщений.

Метод Коада был далек от совершенства, однако являлся простым и по-настоящему объектно-ориентированным. Ниже будет рассмотрен другой подход, OMT, который гораздо сложнее и обладает широкими возможностями, однако с точки зрения объектно-ориентированного подхода имеет более существенные недостатки.

OMT

Метод OMT (Object Modelling Technique) стал одним из наиболее популярных методов объектно-ориентированного анализа. Впервые он был описан в работе Майкла Блага (Michael Blaha), Билла Премерлани (Bill Premerlani), Джеймса Румбаха (James Rumbaugh) и их коллег из компании General Electric [674]. Этот подход в значительной мере основан на традиционных структурных методах и предусматривает использование мощной системы обозначений, которая вместе с тем является достаточно сложной и подробной. Сложность нотации отчасти объясняется тем, что некоторые из поддерживаемых ею средств предназначены для автоматической генерации исходного кода. По существу, язык UML основан на нотации OMT, хотя на него оказали влияние и другие системы обозначений.

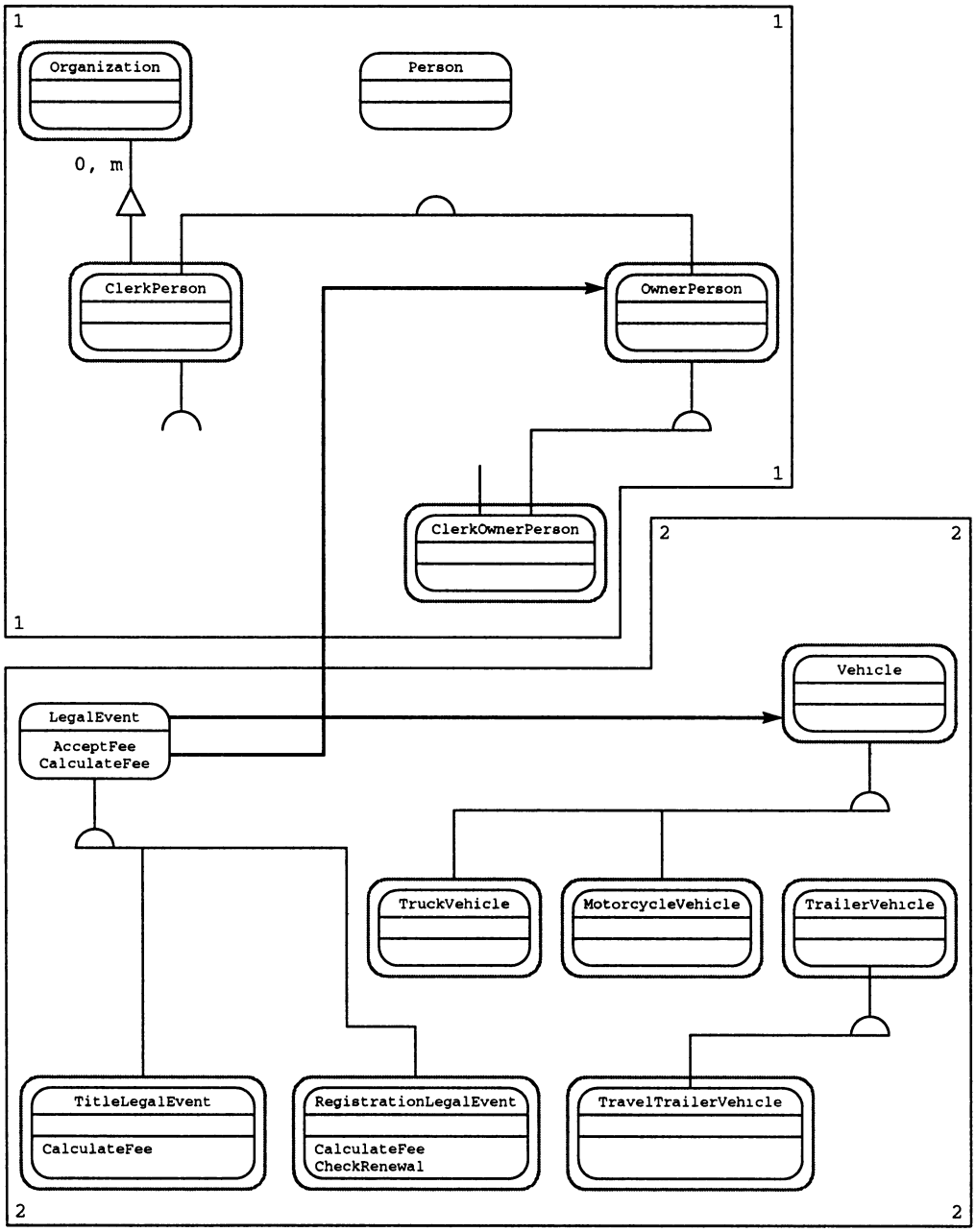


Рис. Б.24. Нотация Коада-Йордона для объектно-ориентированного анализа [171]. (Перепечатано с любезного разрешения издательства Prentice Hall, Upper Saddle River, NJ.)

В рамках метода OMT рассматривается микропроцесс, разделенный на три основных этапа или вида деятельности: анализ, проектирование системы и проектирование объектов. При выполнении **анализа** (первый шаг) предполагается, что существует спецификация требований (сам метод не предлагает способов решения этой задачи). В процессе анализа с использованием трех отдельных систем обозначений создается три разные модели. Вначале строится объектная модель (Object Model), содержащая диаграммы классов и словарь данных. При создании этой модели, по существу, используется система обозначений для моделирования отношений "сущность-связь" (Entity-Relation — ER), а также операции и комментарии, добавляемые к изображениям графических элементов. Далее (второй шаг) для каждого объекта строится динамическая модель (Dynamic Model), в состав которой входят диаграммы переходов из состояния в состояние, созданные в терминах расширенной системы обозначений Харела (Harel), и диаграммы потоков глобальных событий. Третий шаг выполняется лишь в некоторых случаях. При его использовании рассматриваются абстракции самого высокого уровня. На этом шаге строится функциональная модель (Functional Model), которая по назначению ничем не отличается от диаграмм потоков данных. Операции, выявленные в ходе создания динамической и функциональной моделей, добавляются в объектную модель.

Используемые в процессе моделирования объектов условные обозначения представлены на рис. Б. 25—Б. 29. Эта нотация предоставляет большие возможности, чем метод Коада. Обратите внимание на то, что на рис. Б.25 представлены типизированные атрибуты, а для операций указаны списки аргументов и типы возвращаемых значений. Следует также отметить, что углы прямоугольников в изображении экземпляров округлены, а их имена заключены в скобки. Из рис. Б.27 видно, что ассоциации с атрибутами расширяются и "становятся" объектами. Для ассоциаций указываются роли. Кроме того, с ними могут быть связаны квалификаторы. Например, если на рис. Б.26 класс-1 представляет собой каталог, класс-2 — файл, а квалификатором является имя файла, то класс-1 определяет имя файла, обеспечивающее уникальность файла внутри каталога. Таким образом, отношение "один ко многим" между каталогом и файлом преобразуется в отношение "один к одному".

Иерархии классификации показаны на рис. Б.28. Не вызывает никакого сомнения, что для представления функциональных возможностей эта система обозначений предоставляет более широкие возможности, чем нотация Коада. Композиционные структуры представлены на рис. Б.29. Особенно сильная (хотя и не вполне объектно-ориентированная) особенность метода OMT заключается в том, что он позволяет отобразить рекурсивное отношение агрегации (рис. Б.30). На рис. Б.31 можно увидеть всю сложность системы обозначений OMT (хотя даже на этом рисунке она приведена не полностью). Система обозначений, основанная на нотации, приведенной в работе [359], и предназначенная для построения динамической модели, обладает еще более широкими возможностями.

На рис. Б.32 представлена последовательность глобальных событий при выполнении телефонного звонка. Рассматриваемый объект, в данном случае телефонная линия, размещается в центре диаграммы, после чего отслеживается последовательность происходящих событий (рис. Б.32). Таких последовательностей может быть несколько. На основе анализа внешнего поведения можно построить модель внутренних состояний телефонной линии, как продемонстрировано на рис. Б.33. Используемая в данном случае система обозначений основана на диаграммах состояний Харела, что поясняется на рис. Б.34. События, которые могут иметь атрибуты, инициируют действия, которые могут контролироваться условными операторами (предусловиями). При завершении действия объект переходит в требуемое состояние. При переходе в состояние выполняется вид деятельности do. Действия происходят мгновенно, однако для выполнения видов деятельности требуется время.

При переходе системы из данного состояния, выполнение вида деятельности прекращается. В соответствии с нотацией Харела, состояния могут иметь подсостояния и существовать одновременно. Условные обозначения динамической модели представлены на рис. Б.35.

Финальный этап моделирования заключается в построении функциональной модели. Этот этап выполняется далеко не всегда, и в большинстве случаев при этом рассматриваются лишь абстракции достаточно высокого уровня. Подобная деятельность имеет отношение скорее к моделированию процессов (или архитектурному моделированию). В действительности используемая при построении функциональной модели система обозначений почти не отличается от нотации, применяемой для построения диаграмм потоков данных (рис. Б.36). После создания трех (или двух) моделей выявленные операции переносятся в объектную модель, после чего можно переходить к проектированию системы или объектов.

Проектирование системы предусматривает распределение объектов по подсистемам, а также идентификацию параллельных потоков выполнения на основе динамической модели. Подсистемы распределяются среди процессоров и заданий. Кроме того, определяется, будут ли данные храниться в файлах, оперативной памяти или обрабатываться системой управления базой данных. В процессе проектирования необходимо решить вопрос об используемых периферийных устройствах и глобальных ресурсах, выбрать управляющие структуры, задать граничные условия (запуск/завершение/сбой) и приоритеты. Все принятые проектные решения должны быть отражены в соответствующих документах.

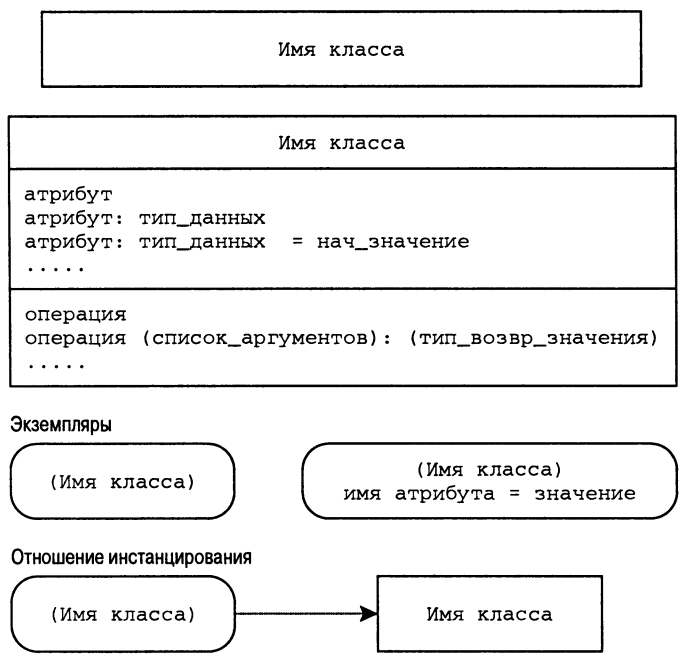


Рис. Б.25. Классы и экземпляры в нотации OMT

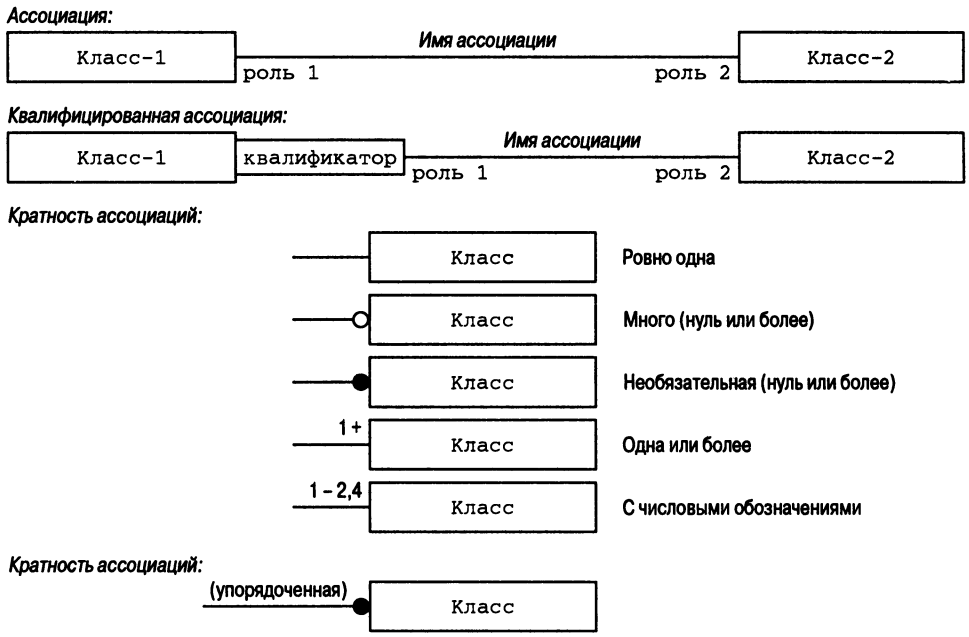


Рис. Б.26. Ассоциации, поддерживаемые в рамках метода OMT

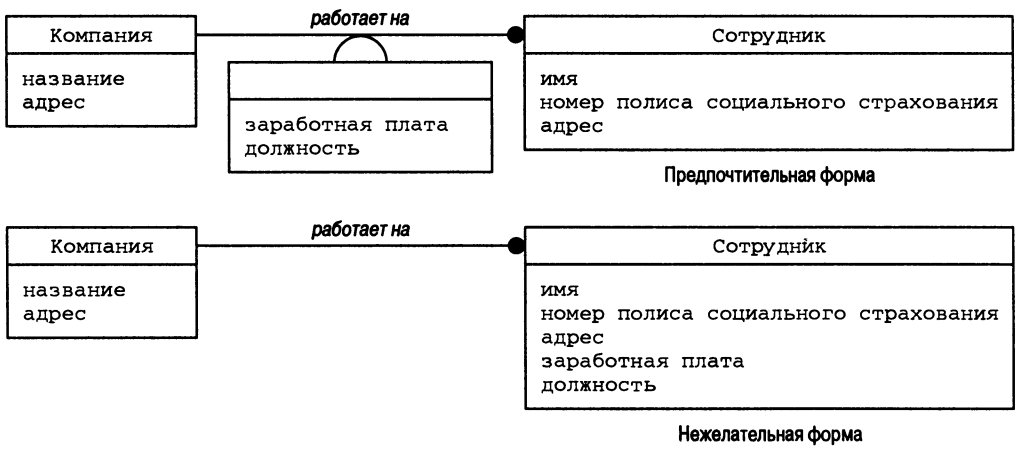


Рис. Б.27. Ассоциации расширяются до объектов, в основном, для добавления атрибутов к уже существующим объектам

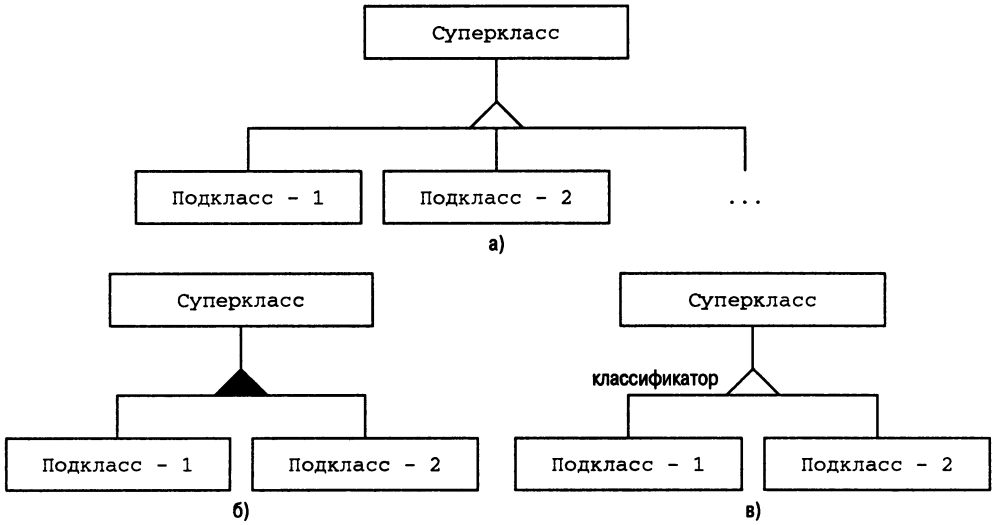


Рис. Б.28. Классификация в методе OMT

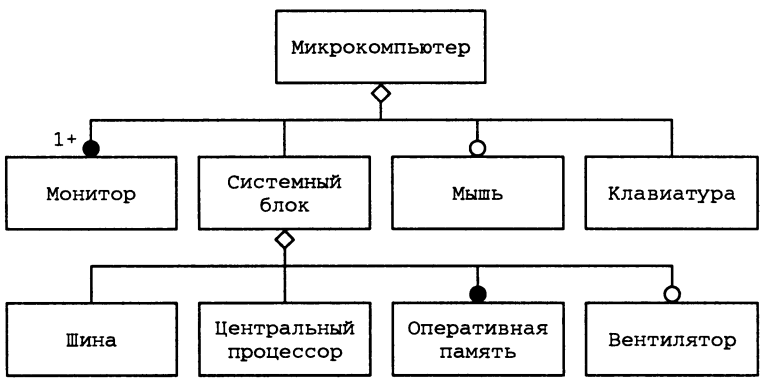


Рис. Б.29. Структура композиции (агрегации) узлов компьютера, представленная средствами метода OMT

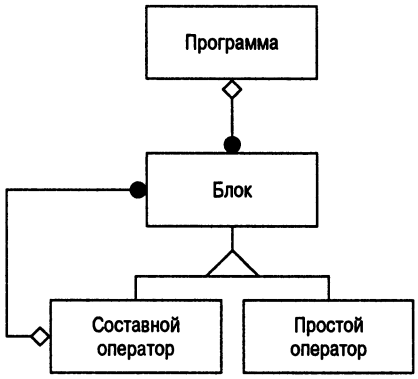


Рис. Б.30. Рекурсивная агрегация, демонстрирующая процесс построения программ

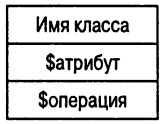
В процессе **проектирования объектов** информация динамической и функциональной моделей преобразуется в операции объектной модели. К оставшимся шагам, которые необходимо выполнить, относятся следующие.

1. Проектирование алгоритмов.
2. Оптимизация путей доступа.
3. Реализация управления.
4. Настройка управляющих структур.
5. Детальное проектирование атрибутов.
6. Построение модулей.
7. Написание отчета о принятых проектных решениях, который включал бы подробное описание объектной, динамической и функциональной моделей.

Метод OMT был предназначен как для анализа, так и для проектирования, хотя при этом предлагался достаточно завершенный подход для анализа, а для проектирования формулировались лишь некоторые полезные эвристики. Метод OMT позволяет решить гораздо больше вопросов, чем подавляющее большинство других методов, однако он остался несовершенным в некоторых определенных областях применения. Кроме того, предлагаемая в рамках этого метода система обозначений очень сложна в изучении и использовании. К его слабым сторонам можно отнести отсутствие возможности высокоуровневого моделирования бизнес-правил и управляющих правил вообще.

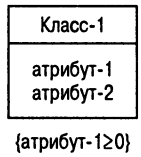
В истории OMT останется популярным методом, поскольку в начале 1990-х годов он учел интересы разработчиков, которые использовали объектно-ориентированный подход и язык C++ и разрабатывали внешние интерфейсы для реляционных баз данных.

Атрибуты и операции классов:

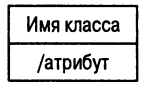


Распространение операций:

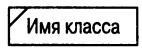
Ограничения, накладываемые на объекты:



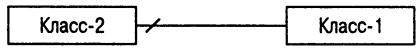
Производный атрибут:



Производный класс:



Производная ассоциация:



Ограничения для ассоциаций:

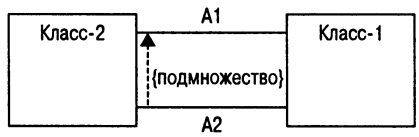


Рис. Б.31. Дополнительные условные обозначения объектной модели метода OMT

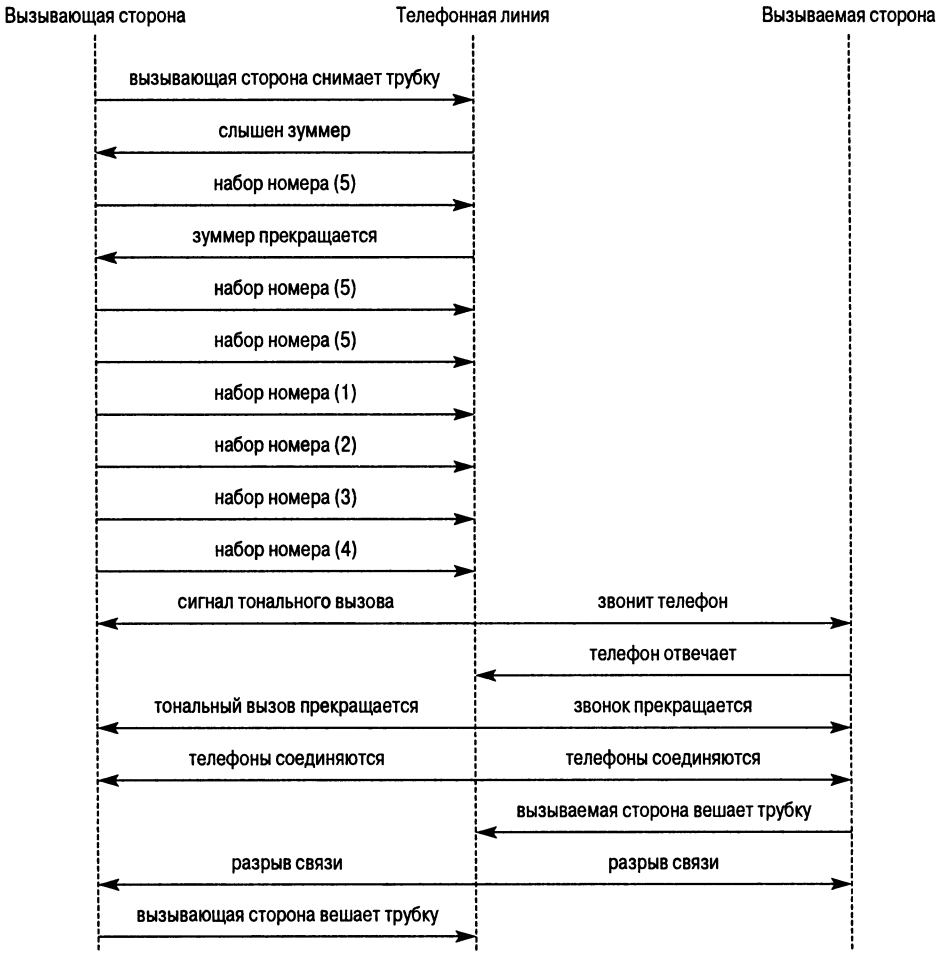


Рис. Б.32. Диаграмма, используемая для отслеживания событий

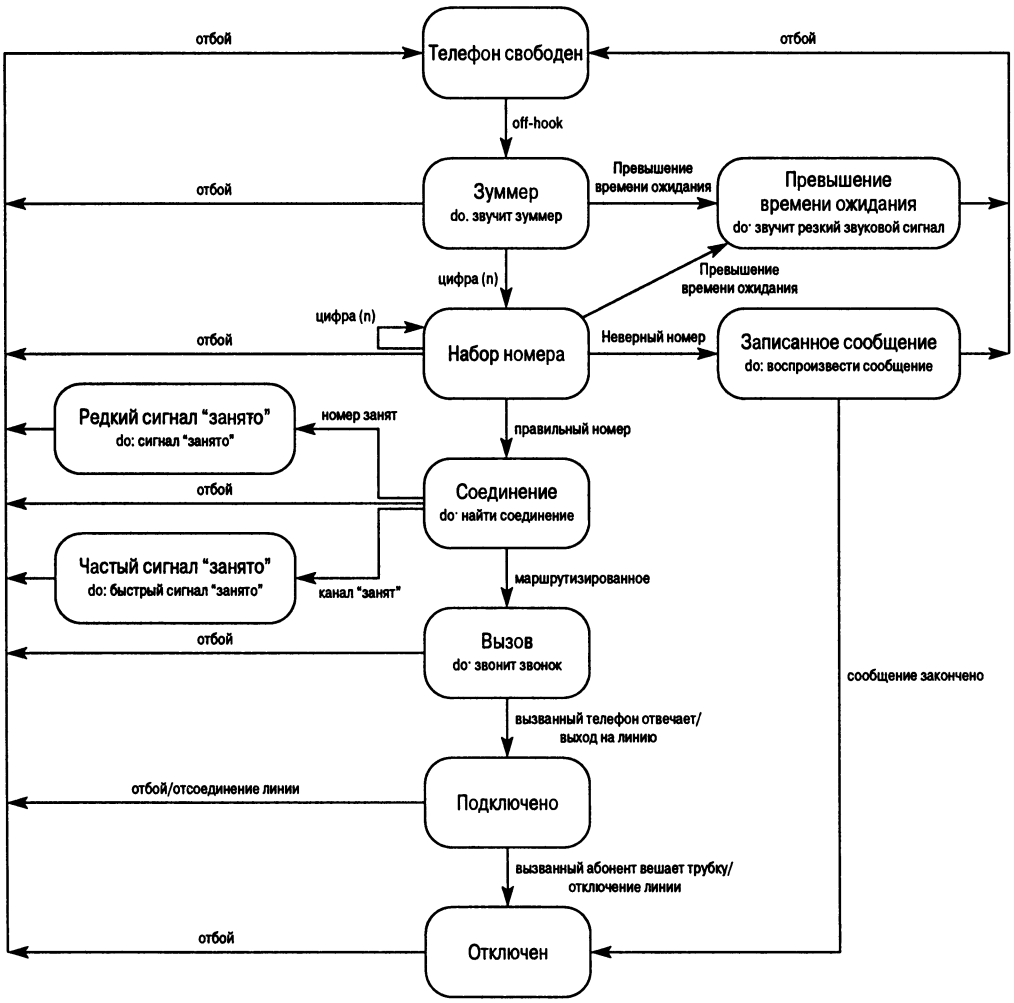


Рис. Б.33. Модель состояний телефонной линии, построенная с использованием метода OMT

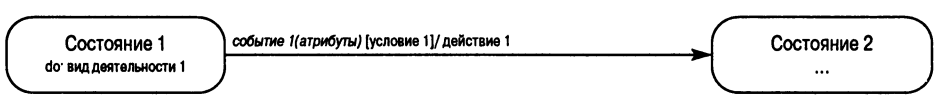


Рис. Б.34. Условные обозначения, используемые в методе OMT для построения диаграммы переходов из одного состояния в другое

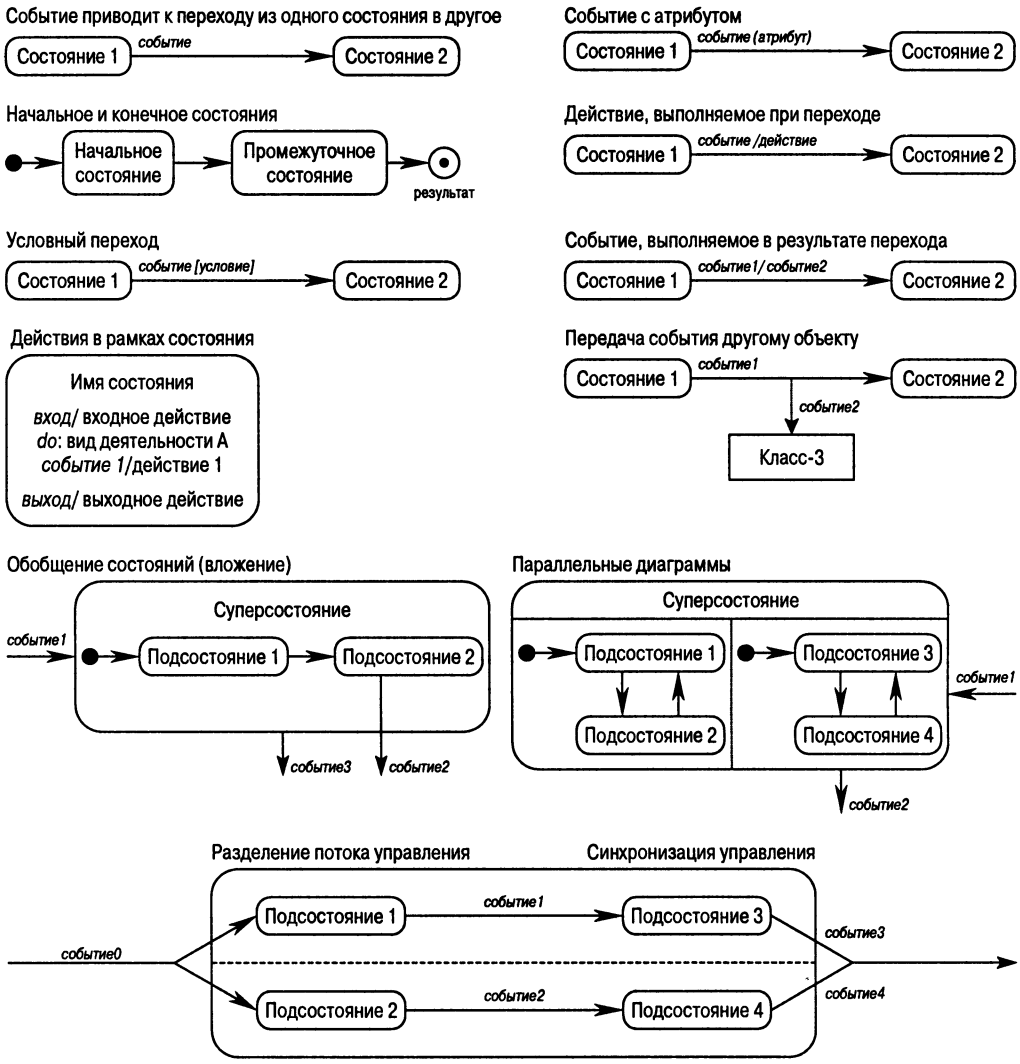
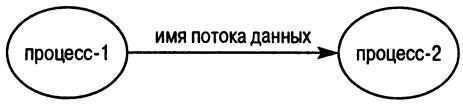


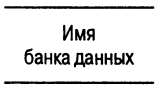
Рис. Б.35. Полная нотация метода OMT для построения динамической модели

Процесс

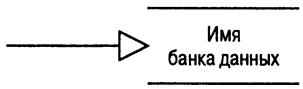
Потоки данных между процессами



Банк данных или файловый объект

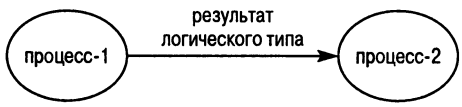


Поток данных,
в результате которого получается банк данных

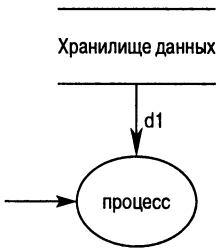


Объекты-исполнители
(выступающие в качестве источника или приемника данных)

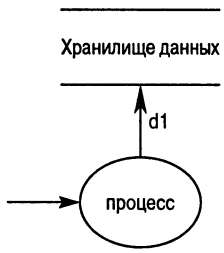
Поток управления



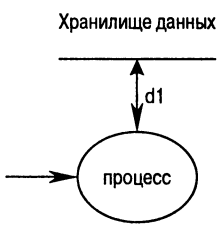
Доступ к значению из хранилища данных



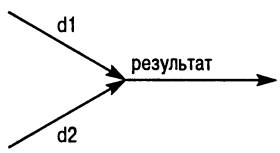
Обновление значения в хранилище данных



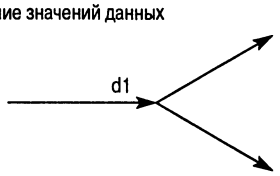
Доступ к значению в банке данных и его обновление



Композиция значений данных



Дублирование значений данных



Декомпозиция значений данных

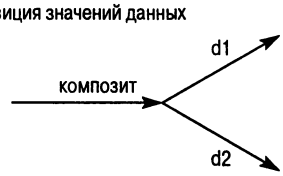


Рис. Б.36. Полная нотация метода OMT для построения функциональной модели

Метод Мартина-Оделла и средства проектирования Ptech

Средства Ptech от компании Associative Design Technology представляют собой набор методов и инструментов, с использованием которых можно выполнять как анализ, так и проектирование. Этот комплект инструментов обладает некоторыми свойствами, характерными для объектно-ориентированных методов. Кроме того, разработанное компанией ADT CASE-средство позволяет также генерировать код на языках C++ и Ontos.

В основу метода Ptech положена идея разработки систем посредством их сборки из повторно используемых компонентов. Конечно, при таком подходе анализ и логическое проектирование отделяются от реализации. Поэтому в рамках этого метода основное внимание уделяется скорее не тому, что делается, а тому, как это делается. В этом смысле можно сказать, что метод Ptech ориентирован на процесс, а не на строгое соответствие объектно-ориентированной технологии, хотя оба подхода имеют много общего. В методе Ptech представление, направленное на процесс, комбинируется с абстрактными концепциями объектно-ориентированного проектирования, которые больше сконцентрированы на данных, а также с некоторыми идеями теории множеств и искусственного интеллекта.

Эдвардс, автор метода Ptech, подчеркивает важность существования формальной основы для любого объектно-ориентированного метода и критикует так называемые "наивные" подходы к объектно-ориентированному анализу, например методы Коада-Йордона или OMT [247]. В рамках метода Ptech основное внимание уделяется прототипированию. Эдвардс полагает также, что рабочие спецификации абсолютно необходимы для обратного проектирования и поддержки. И с этим трудно не согласиться.

Метод Ptech во многом послужил основой для разработки подхода, описанного в книге [521]. Метод Мартина-Оделла стал главным источником идей, которые были использованы для разработки нотации диаграмм видов деятельности языка UML.

В рамках метода Ptech рассматриваются диаграммы трех видов: концептуальные диаграммы (в общих чертах эквивалентны расширенным диаграммам "сущность-связь" в стиле Бачмена (Bachman)), называемые также схемами объектов или концептуальными схемами; диаграммы или схемы событий (играют роль диаграмм переходов из состояния в состояние); и диаграммы видов деятельности/функций (аналогичны диаграммам потоков данных). Первых два типа диаграмм поддерживаются формальными языками: исчислением классов и исчислением событий. Основные архитектурные компоненты метода Ptech представлены на рис. Б.37.

На концептуальных диаграммах отображаются статические аспекты процессов, а диаграммы событий позволяют отразить их динамику. При совместном использовании концептуальных диаграмм и диаграмм событий можно представить создание и уничтожение экземпляров. Диаграммы видов деятельности предоставляют высокоуровневое функциональное представление, которое во многом аналогично диаграммам потоков данных. При этом понятия рассматриваются как множества, а для их описания применяется теория множеств и такие простые теоретико-множественные операции, как объединение, пересечение и разность.

В методе Ptech наследование изображается с помощью условных обозначений, показанных на рис. Б.38. На приведенной диаграмме средством платежа может являться либо жетон, либо монета номиналом в 5, 10 или 50 пенсов и ничего больше. Эти четыре подтипа сегментируют понятие "средство платежа". Последние три из них формируют разбиение понятия "монета". Значение стоимости может быть представлено монетой или другой, не определенной на данный момент сущностью, что иллюстрируется двойной линией в нижней части контура класса. В методе Ptech события изображаются в виде прямоугольников с округленными

углами. Система обозначений, представленная на рис. Б.39, определена в версии метода Ptech, предназначенной для исследования семантики данных. Эта модификация метода здесь рассматриваться не будет. На рис. Б.39–Б.41 представлены фрагменты диаграмм, построенных в процессе проектирования торгового автомата.

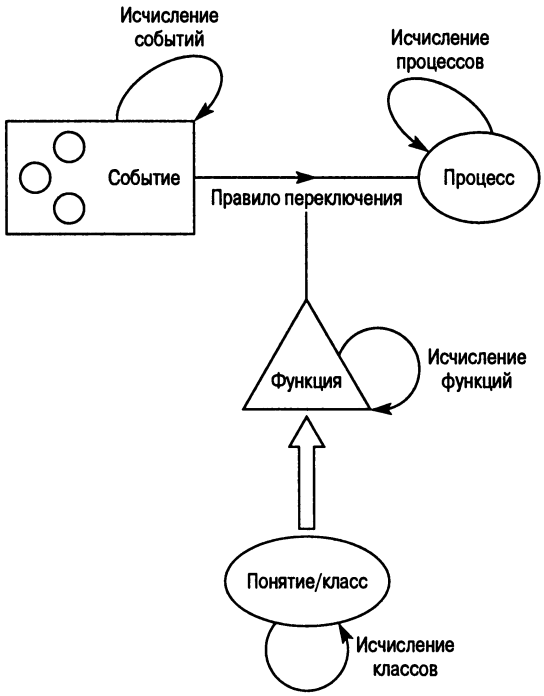


Рис. Б.37. Представление процесса разработки в методе Ptech

События являются *n*-местными понятиями (т.е. объектами с *n* атрибутами), с которыми связаны определенные пред- и постусловия. Постусловия означают, что каждый процесс связан с достижением определенной цели. Для событий теоретико-множественное исчисление классов рассматривается параллельно с функциональным исчислением, которое основано на лямбда-исчислении. Это позволяет задать условия переключения и бизнес-правила. На рис. Б.39 представлена диаграмма событий, построенная в процессе проектирования торгового автомата. Прямоугольники обозначают события, а стрелки, заканчивающиеся кружками, представляют правила переключения или предусловия. Треугольники задают постусловия или моменты принятия решений. Постусловия могут принимать значение true (истина) или false (ложь), в зависимости от результатов завершения предыдущих процессов (которым соответствуют входящие стрелки).

Вообще говоря, нотация метода Ptech для моделирования событий представляет собой систему обозначений конечного автомата с расширенной семантикой, которая позволяет описывать ограничения, предусловия и т.д. Вне всякого сомнения, такое представление является чрезвычайно полезным. Однако согласование таких диаграмм с объектной моделью, в процессе которого пиктограммы событий связываются стрелками с условными обозначениями

классов на отдельной объектной диаграмме, оказывается не очень удобным, а иногда и чрезвычайно беспорядочным. Основной недостаток рассматриваемого подхода характеризуется следующей фразой: "Возможность повторного использования операции является ключевым фактором в объектно-ориентированной спецификации и реализации". С этим нельзя согласиться. Ключевой является возможность повторного использования типов. Именно поэтому такое огромное значение имеет объединение объектных и поведенческих моделей. Диаграммы событий метода Ptech особенно полезны для исследования управляющей структуры системы. Они содержат много полезной информации, чего недостает во многих других объектно-ориентированных подходах. Однако рассмотрение понятий в виде множеств нарушает сам дух объектно-ориентированной технологии, поскольку понятия с методами являются намного большим, чем просто множества. Метод Ptech фокусирует основное внимание на событиях, а не на объектах. В этом заключается и его сила, и слабость, поскольку истинные объектно-ориентированные методы фокусируются на повторно используемых объектах и обычно оказываются недостаточно развитыми в вопросах моделирования глобальной динамики. Однако в методе Ptech события представляют собой объекты, которые можно использовать в иерархиях классификации. Поведенческая модель Ptech тщательно продумана, более проста для понимания и не такая детальная, как соответствующая модель метода OMT.

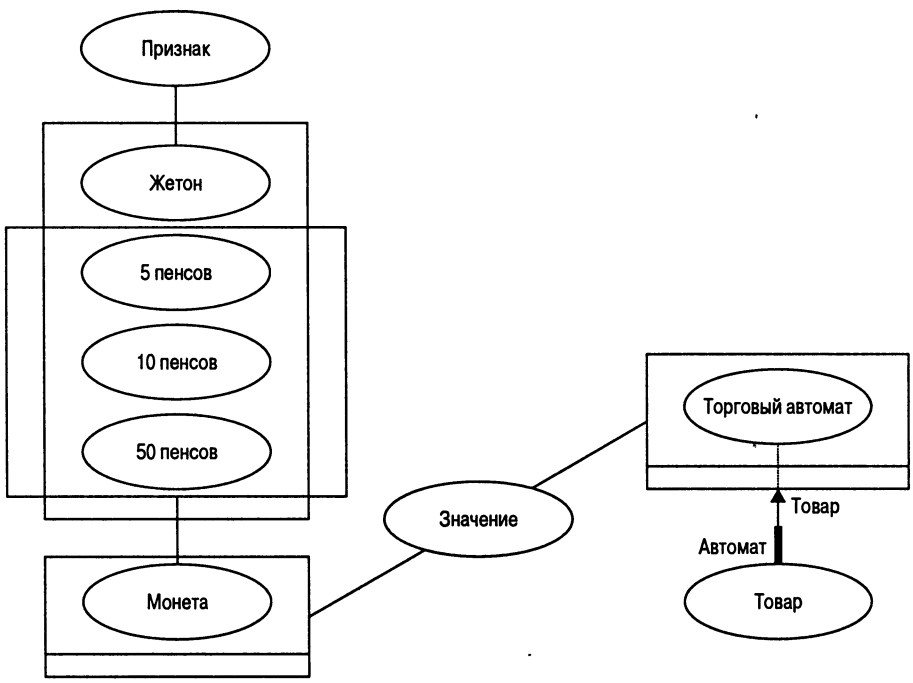


Рис. Б.38. Представление наследования на концептуальной диаграмме метода Ptech

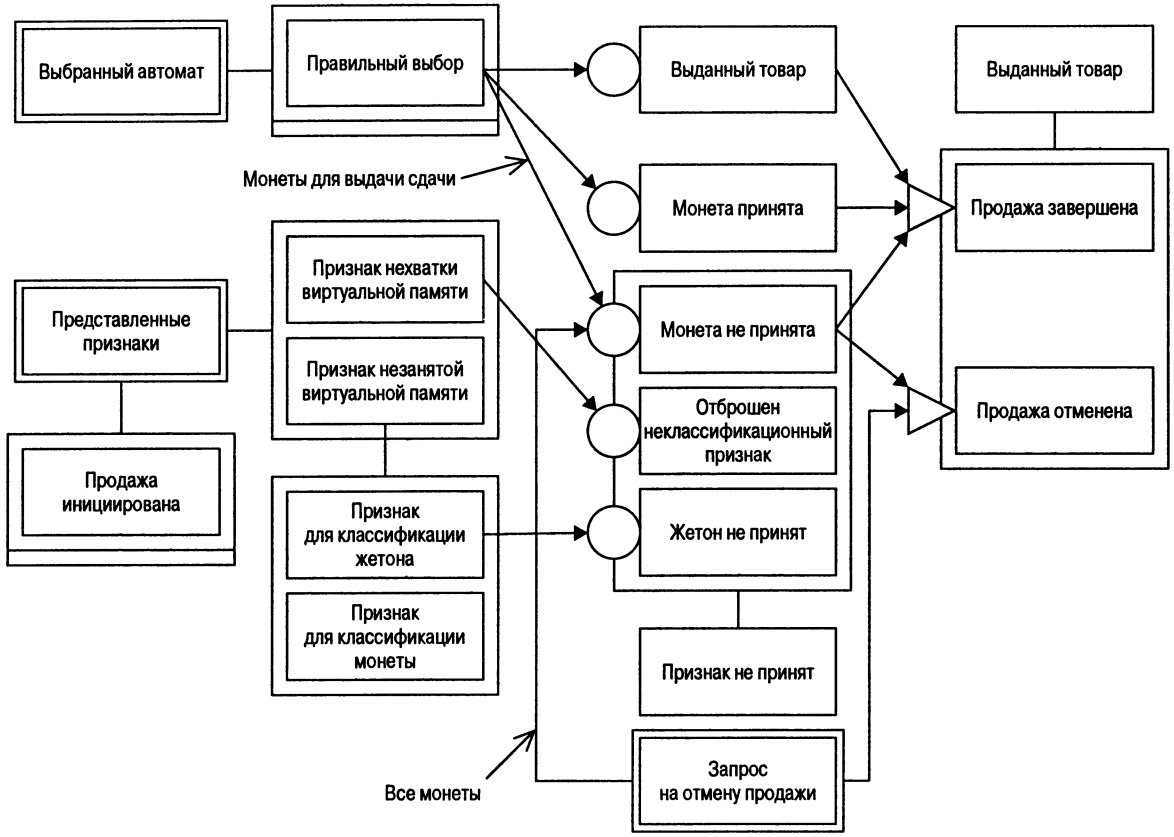


Рис. Б.39. Диаграмма событий в методе Ptech

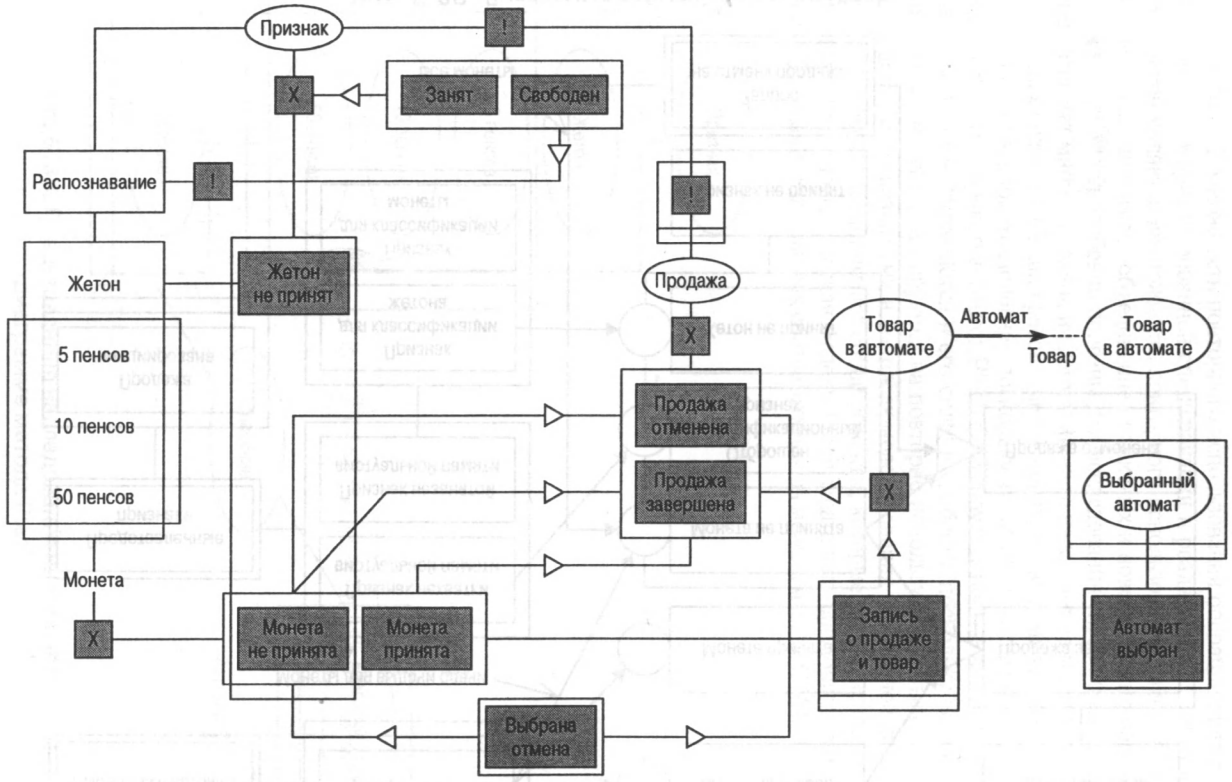


Рис. Б.40. Комбинированная диаграмма понятий и событий, построенная для торгового автомата

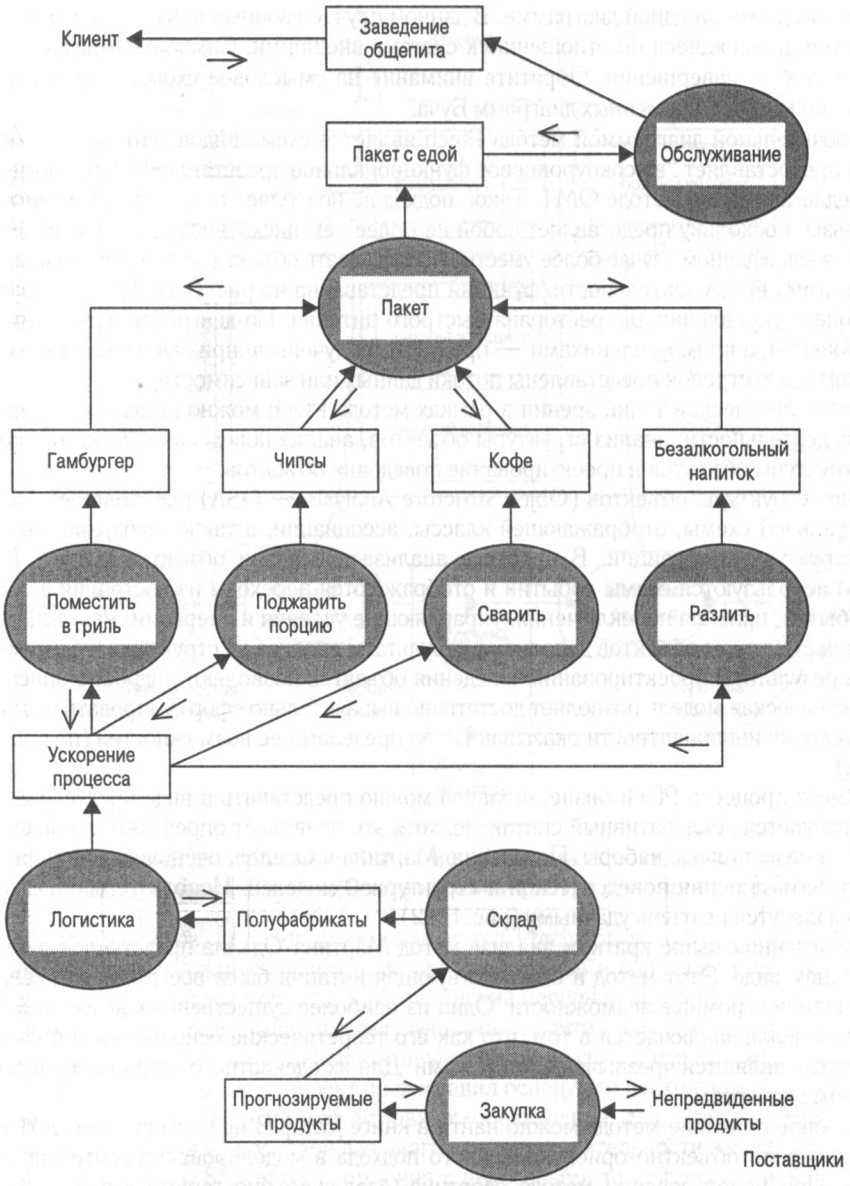


Рис. Б.41. Диаграмма видов деятельности/функций для системы из предметной области общепита

Как видно из рис. Б.40, концептуальные диаграммы и диаграммы событий метода Ptech можно объединить на одной диаграмме. В данном случае двойные прямоугольники обозначают события, являющиеся по отношению к системе внешними. Символ ! означает операцию создания, а X — завершения. Обратите внимание на смысловое сходство этих символов с обозначениями ! и * временны*х диаграмм Буча.

Заключительной диаграммой метода Ptech является схема видов деятельности/функций, которая предоставляет "высокоуровневое функциональное представление". Аналогичные модели предлагаются и в методе OMT. Такой подход не позволяет получить какой-либо ощутимой пользы, поскольку представляет собой не более чем нисходящую модульную декомпозицию, тогда как в данном случае более уместно использовать объектно-ориентированный подход.

Диаграмма видов деятельности/функций представлена на рис. Б.41. На ней иллюстрируется процесс управления для ресторана быстрого питания. На диаграмме кружками обозначены процессы, а прямоугольниками — продукты, полученные при выполнении этих процессов. С помощью стрелок представлены потоки данных или зависимости.

С методологической точки зрения в рамках метода Ptech можно выделить четыре основных вида деятельности: анализ структуры объектов, анализ поведения объектов, проектирование структуры объектов и проектирование поведения объектов.

Анализ структуры объектов (Objet Structure Analysis — OSA) предполагает построение концептуальной схемы, отображающей классы, ассоциации, а также иерархии классификации и структуру композиции. В процессе анализа поведения объектов (Object Behaviour Analysis) используются схемы событий и отображаются переходы из состояния в состояние, типы событий, правила переключения, управляющие условия и операции. Результаты проектирования структуры объектов дополняют результаты анализа их структуры нюансами реализации, а результаты проектирования поведения объектов позволяют подробно описать методы. Поведенческая модель позволяет достаточно выразительно сформулировать предусловия, однако условия инвариантности оказываются за пределами ее возможностей (по крайней мере, пока).

В рамках процесса Ptech бизнес-правила можно представить в виде предусловий. Кроме того, допускается декларативный синтаксис, хотя это позволяет определять лишь единичные правила, а не их полные наборы. По мнению Мартина и Оделла, очень важной является возможность объединения поведенческой и структурной моделей. Используемые для этого соглашения кажутся не очень удачными (рис. Б.42).

В приведенном выше кратком анализе метод Мартина-Оделла представлен лишь в наиболее общем виде. Этот метод и соответствующая нотация были весьма выразительными и предоставляли огромные возможности. Один из наиболее существенных недостатков метода Мартина-Оделла заключается в том, что как его теоретические основы, так и практические инструменты являются чрезвычайно сложными. Для их адекватного применения нужно быть настоящим экспертом.

Подробное описание метода можно найти в книге [521]. В ней содержатся глубокие рассуждения о роли объектно-ориентированного подхода в моделировании деятельности предприятий. При использовании метода Мартина-Оделла можно гарантировать, что бизнес-правила и ограничения будут учтены еще в процессе моделирования предметной области.

Метод Ptech с успехом применялся Государственной службой здравоохранения Великобритании для моделирования в области клинической медицины. Однако при этом стало понятно, что в метод необходимо внести некоторые изменения и расширения [284].

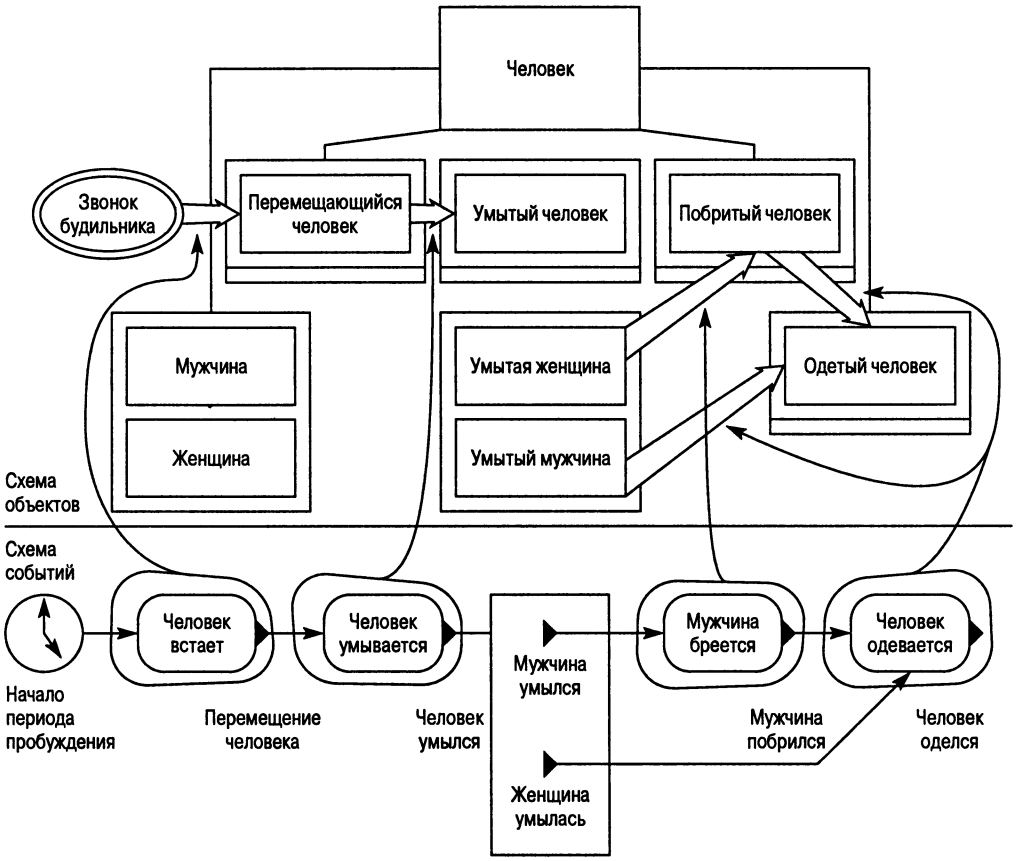


Рис. Б.42. Диаграмма событий и диаграмма объектов (понятий) с перекрестными ссылками. (Взято из книги [521] с разрешения авторов.)

Группы специалистов, которые применяли метод Ptech, были небольшими и, что более интересно, обычно состояли из одного или двух аналитиков и одного или двух клинических специалистов (врачей, медсестер и т.д.). Медицинский персонал обучили методике анализа, и через некоторое время они стали самыми лучшими аналитиками. Это оказалось возможно благодаря тому, что объектно-ориентированный подход позволял строить модели реального мира из области медицины. Этот проект предоставил основу для нескольких моделей анализа, описанных в работе [282]. Система условных обозначений для представления событий (рис. Б.40 и Б.42) послужила основой для диаграмм видов деятельности языка UML, которые рассматривались в главе 8. Для ознакомления с некоторыми теоретическими аспектами метода Мартина-Оделла можно познакомиться с работами [329, 523, 524].

Метод "класс-связь"

В работе [225] описывается метод "класс-связь" для объектно-ориентированного анализа и проектирования, который распространялся французской компанией Softeam. Этот метод был специально предназначен для выполнения проектов на языке C++. Его основная особенность

заключается в использовании системы обозначений, основанной на моделях "сущность-связь" Чена (Chen) и, следовательно, потенциально совместимой со структурным методом Merise, который в то время был широко распространен во Франции. Необходимо отметить также то, что в методе "класс-связь" использовались формальные методы, основанные на Eiffel-подобных утверждениях. Кроме того, этот метод поддерживался также CASE-средством Objecteering.

В рамках метода "класс-связь" определено три различные модели: модель "объект/сущность", модель переходов из состояния в состояние и модель потоков данных. Этот метод заслуживал внимания, поскольку позволял выполнять полноценный объектно-ориентированный анализ, который не уступал методу OMT. Как и любые другие подходы, этот метод можно усовершенствовать, например, уточнив понятие "абстрактного отношения". Поддерживаемая модель потоков данных позволяет определить связь между процессами, использует концепцию событий и обеспечивает возможность их упорядочения. Последовательность действий, выполняемых в рамках метода "класс-связь", аналогична подходу OMT: создается модель данных/классов; формулируются утверждения; для описания событий и методов строится динамическая модель; и наконец, при необходимости создаются диаграммы потоков данных. В этом методе понятию инкапсуляции уделяется значительно больше внимания, чем в методе OMT. Десфрей считает, что модель переходов из состояния в состояние можно улучшить, введя специальные состояния и определив механизм синтеза состояний.

С точки зрения возможностей метода следует заметить, что, кроме всех хорошо известных структурных компонентов моделирования данных, его модель обеспечивает полную поддержку наследования. Наряду с тем, что отношения можно трактовать как объекты, Десфрей делает важное разграничение между классами и отношениями: "Отношения не могут содержать методы. Их уникальная функция состоит в том, чтобы связывать классы". После некоторых размышлений можно прийти к выводу, что это обоснованное разграничение, которое ранее в этой области никем не было сделано. Хотя в методе OMT специально для этой цели можно использовать "операции связи", нелегко найти для них должного применения. Это облегчает решение задачи разделения атрибутов и классов. Отношение представляет собой направленную связь между двумя классами и позволяет одному классу "знать" об экземплярах другого класса. Двунаправленные отношения отбрасываются. Десфрей также утверждает, что "отношение является частью определения понятия, которое представляет класс". Еще одним новшеством можно считать появление наследуемых инвариантов класса, соответствующих бизнес-правилам в некоторой ограниченной форме. В рамках метода считается, что классы сформированы с использованием композиции или агрегации их атрибутов, рассматриваемых в качестве типов объектов. Общее правило моделирования сущностей заключается в том, что только разрешенные атрибуты могут принадлежать базовому классу. Это позволяет избежать конфликтов, которые возникают при использовании множественного наследования и в некоторых других случаях. В методе Десфрея какой-либо элемент не может принадлежать сразу двум классам. Строение классов описывается *схемами*, соответствующими предметной области в терминах Коада-Йордона. Поэтому классы также "страдают" от недостаточно продуманной семантики, поскольку они не могут обмениваться сообщениями. "Схема — это представление предметной области. Она состоит из групп классов. Она составляет описание предметной области. Класс принадлежит одной и только одной схеме. Таким образом, приложение можно разделить в соответствии со схемой" (стр. 86). "Метод "класс-связь" допускает наличие отношений [взаимного] использования (передачу сообщений) между классами, однако не допускает их передачи между схемами" (стр. 91). Другими словами, в методе Десфрея обеспечивается, чтобы классы были частью одной и той же схемы. Для улучшения

этой ситуации определенные классы схемы могут быть объявлены как "интерфейсные". Кроме того, схемы обладают инвариантами, которые определяются как объединение всех инвариантов, содержащихся в классах, плюс "некоторые общие выражения" (стр. 93). Другими словами, отношение взаимного использования определяется следующим образом: если классы `C1` и `C2` используют службы друг друга, то связь между ними является взаимной. А это возможно только внутри схемы. Однако если схема S2 может использовать схему S1, то схема S1 не должна использовать схему S2. Десфрей дает некоторые указания для устранения взаимных связей на уровне схем. Отношения использования между классами подразделяются на три следующие категории.

1. *Операционное* использование. Класс `C1` использует класс `C2` лишь для того, чтобы позволить методу класса `C1` завершить выполнение операции.
2. Использование *в контексте*. Класс `C1` использует класс `C2` в качестве параметра метода.
3. *Типичное* использование. Класс `C1` использует `C2`, если `C1` наследуется от `C2`. Схема S1 связана отношением использования со схемой S2, если хотя бы один из классов схемы S1 использует хотя бы один из классов схемы S2. Классы схемы S1 могут иметь доступ только к интерфейсным классам схемы S2, но не к "внутренним" классам. Поэтому схемы напоминают "оболочки" метода SOMA. Кроме того, схемы можно связать при помощи наследования.

Наиболее важное новшество метода "класс-связь" заключается в использовании утверждений, хотя и в ограниченной форме предусловий и постусловий. Вместе с тем в рамках этого метода поддерживаются инварианты классов.

OSA

Подход OSA (Object-Oriented Systems Analysis) был предложен компанией Hewlett Packard. Этот метод схож с методом OMT в том смысле, что в нем анализ также разделяется на три отдельных (но взаимосвязанных) вида деятельности. Причем при выполнении каждого из них используется собственная система обозначений, хотя и более простая. Однако вместе с этим имеется и ряд различий [257].

В рамках метода OSA сначала строится модель отношений между объектами (ORM — Object Relationship Model), для чего используется нотация "сущность-связь". В этой модели описываются атрибуты, иерархия классификации и отношения агрегации, а также представляется семантика данных в терминах ассоциаций. Более общие ограничения просто записываются на диаграммах в виде комментариев, располагаемых рядом с объектом, к которому они относятся. Таким образом, ограничения не связаны с наследованием. По утверждению авторов метода, его преимущество заключается в том, что он не накладывает никаких ограничений на специалиста по анализу. Однако в работе [257] говорится следующее: "Поскольку комментарии не накладывают ограничений на наборы классов объектов или отношений, диаграммы ORM имеют один и тот же смысл как с комментариями, так и без них". Это означает, что в конечной модели информация теряется. Полная система обозначений для построения модели ORM представлена на рис. Б.43.

Система обозначений, используемая для представления переходов из состояния в состояние, позволяет описать поведение каждого объекта. В модели поведения объектов (Object Behaviour Model — OBM) описываются методы объекта, однако модель ORM нельзя расширить для

включения подобной информации. Все триггеры, ограничения реального времени, исключения и события обрабатываются в рамках модели OBM. Именно в этом наиболее четко просматривается сходство подхода OSA с методом OMT, хотя в данном случае основное внимание уделяется моделированию "внутреннего" поведения объектов, а не взаимодействию между ними. На рис. Б.44 представлена система обозначений, которую можно использовать для построения модели OBM.

И наконец, передача сообщений описывается моделью взаимодействия объектов (Object Interaction Model — OIM), которая несколько напоминает диаграмму потоков данных. Преимущество метода OSA, по сравнению с OMT, заключается в том, что в рамках модели OIM более явно представляется процесс передачи сообщений. В определенном смысле в модели взаимодействия объектов представление потоков данных сочетается с отображением переходов из состояния в состояние. На рис. Б.45 представлены условные обозначения, с использованием которых можно построить модель OIM.

Правила декомпозиции определены для построения как модели ORM, так и OBM. Некоторые используемые в методе OSA условные обозначения выглядят слишком сложными, однако предлагаемые в нем идеи, вне всякого сомнения, оказались весьма плодотворными. Каждую из моделей метода OSA — ORM, OBM и OIM — можно представить на различном уровне абстракции. Соответствующие условные обозначения показаны на рис. Б.46. На приведенных в работе [257] подробных диаграммах содержится большое количество дополнительной информации, связанной с ограничениями, исключениями и приоритетами.

Система обозначений модели OIM является, вероятно, наиболее полезной. Однако она отличается от всех остальных методов объектно-ориентированного анализа, поэтому ее нелегко распространить на другие технологии. Некоторые возможности, обеспечиваемые использованием динамической модели метода OMT, можно реализовать и при помощи методики OSA.

Метод OSA основан на модели, т.е. аналитикам не нужно придерживаться строго определенной последовательности шагов, а достаточно просто создать модель. Как и в случае с методами Ptech и OMT, остается нерешенным вопрос, каким образом результаты трех независимых вариантов анализа должны объединяться в единственной модели объектов, хотя вопросам согласованности и совместимости диаграмм некоторое внимание все же уделяется. Утверждается, что метод OSA не делает различия между классами и атрибутами, благодаря чему достигается совместимость. Это утверждение не учитывает глубокой философской значимости проблемы объектов/атрибутов. В методе OSA предусмотрены обозначения для множественного наследования, однако он, как и метод Коада-Йордона, исключает какую бы то ни было процедуру разрешения конфликтов.

В приложении книги [257] содержится изложение формальных логических принципов, лежащих в основе модели отношений объектов — компонента метода OSA. Вызывает небольшое беспокойство тот факт, что представленная объектно-ориентированная модель, выполненная на языке первого порядка, не использует одно из наиболее простых свойств объектно-ориентированного подхода — возможность описывать сложные структуры непосредственно в терминах более высокого порядка.

OSA представляет собой довольно сложный метод, который выдерживает сравнение с методом OMT и имеет, на мой взгляд, чуть более хорошую систему обозначений. (Уровни представлений в методе OSA представлены на рис. Б.46.) Однако популярность метода OMT и отсутствие CASE-средств, поддерживающих метод OSA, не позволили ему добиться большого успеха.

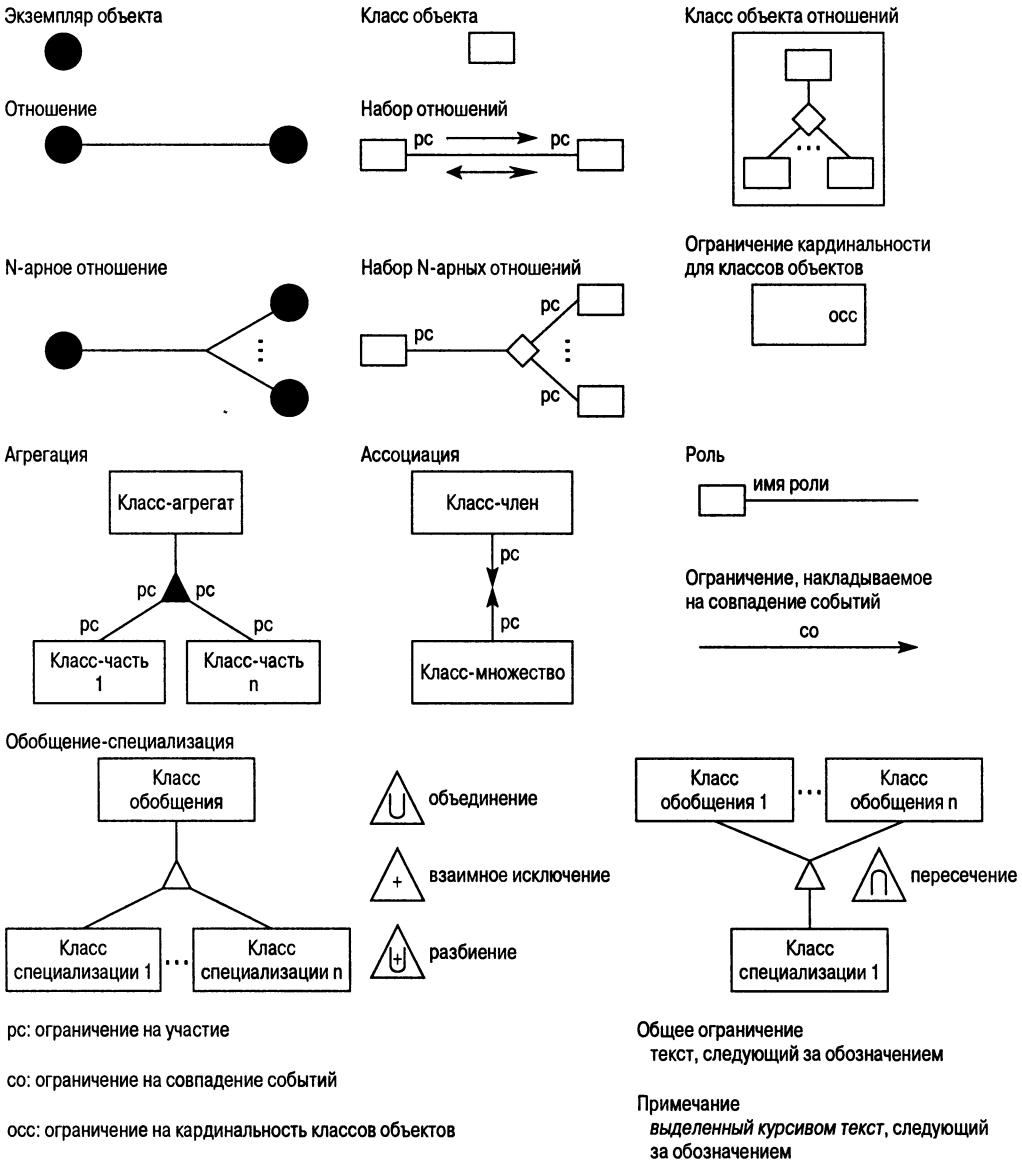


Рис. Б.43. Модель отношений объектов, выполненная в соответствии с методом OSA. (Адаптированный вариант модели, представленной в книге [257].)

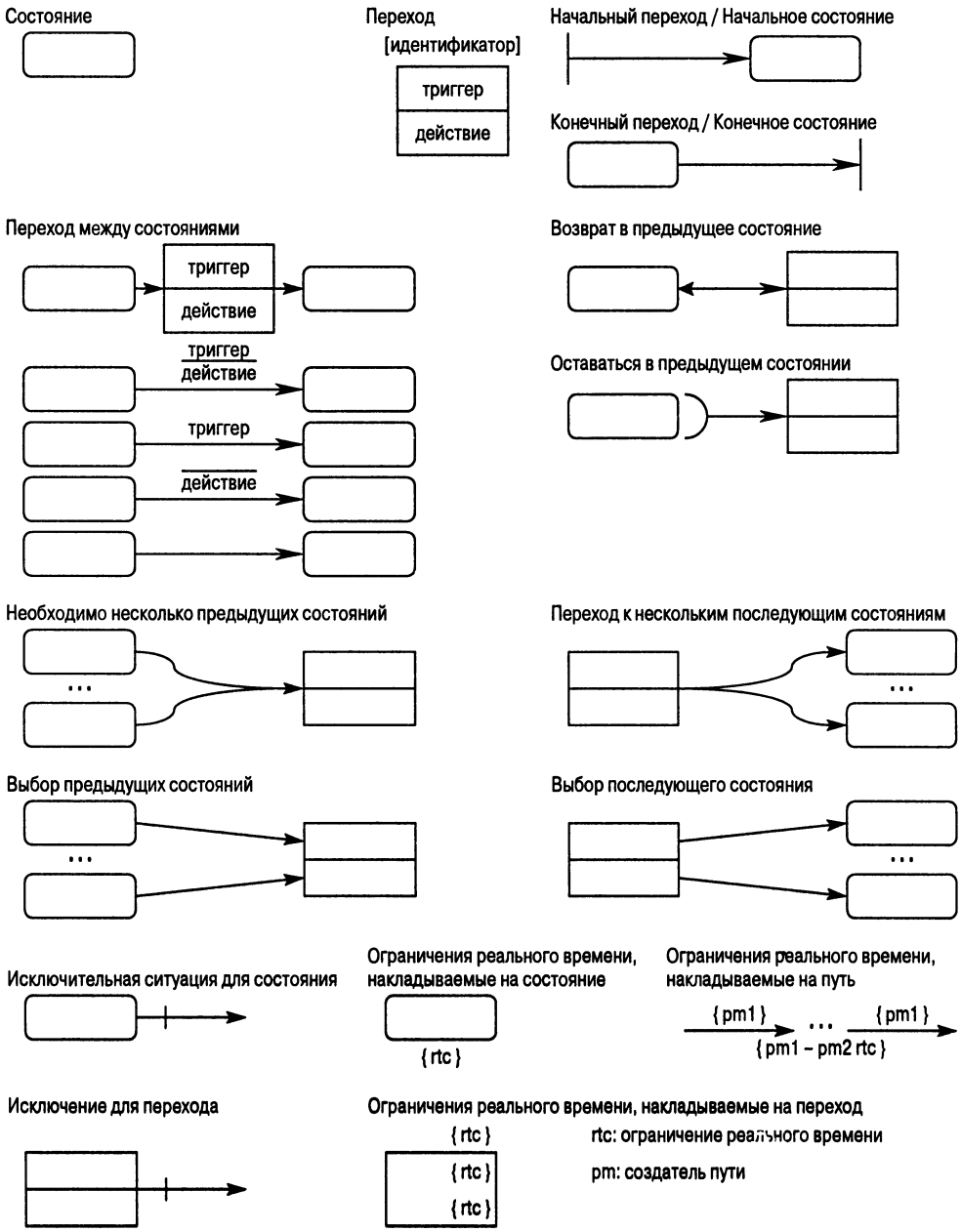


Рис. Б.44. Модель поведения объектов в системе обозначений метода OSA

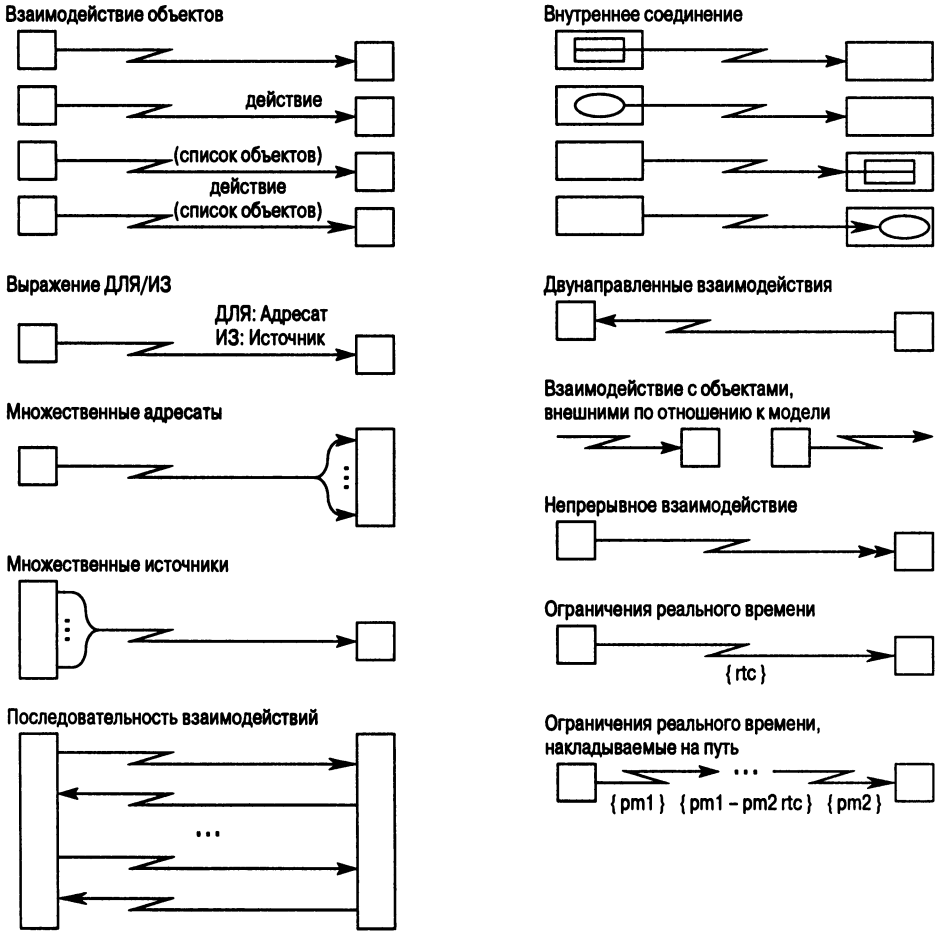


Рис. Б.45. Модель взаимодействия объектов в методе OSA

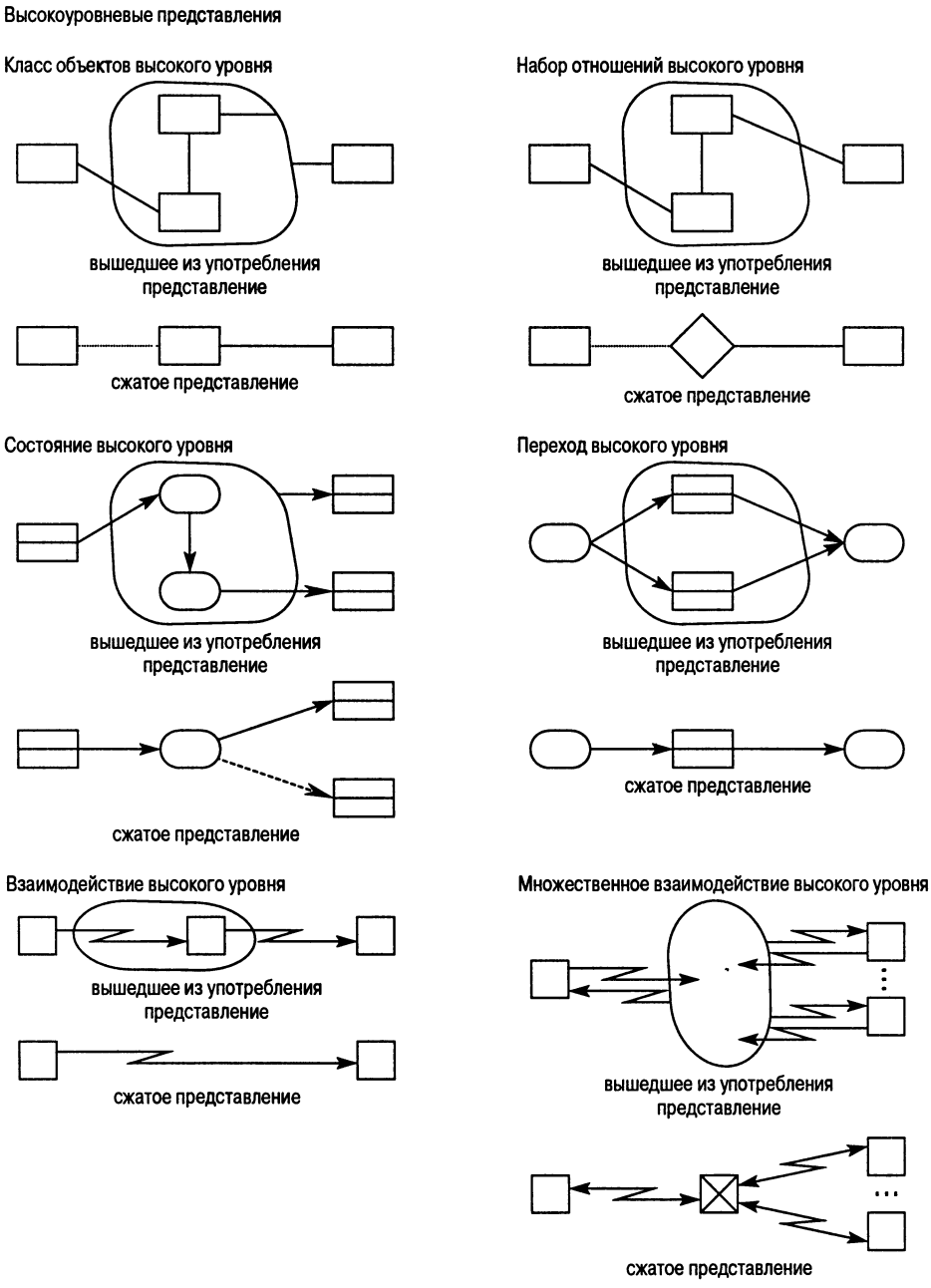


Рис. Б.46. Уровни представлений в методе OSA

Метод SEOO

Рассмотрим метод SEOO (System Engineering OO), который был разработан Британской компанией LBMS. Этот метод включает четыре аспекта.

1. Методы структуризации работ.
2. Метод моделирования совместно используемых объектов.
3. Способы проектирования специальных графических пользовательских интерфейсов [652].
4. Принципы связывания реляционных баз данных.

Метод заключается в построении модели процесса на основе определенных технических приемов. Предлагаемый жизненный цикл является циклом быстрой разработки, при котором основное внимание уделяется завершающим стадиям получения исполняемого кода, который может быть оценен пользователями.

Этот процесс имеет некоторые общие свойства с методом OPEN [334], а посему был объединен с ним, а также с шаблонами процесса Catalysis в объединенном документе, представленном рабочей группе по развитию стандартов объектного программирования (OMG) в 1999 году. SEOO представлял собой один из первых методов, в котором друг от друга отделялись объекты предметной области, объекты приложений и интерфейсные объекты.

В работе [137] показано, что структурные методы, предназначенные для моделирования данных в системах обработки данных, могут использоваться в качестве высокоэффективных средств анализа и проектирования. Однако при создании представления теряются многие семантические возможности, доступные аналитику. Более того, в [137] утверждается, что при разделении данных и процессов теряются преимущества многих технологий объектно-ориентированного анализа, и предлагается способ моделирования совместно используемых объектов. Этот метод создавался как объектно-ориентированный, но все еще сохранял преимущества методик моделирования данных и, кроме того, был пригоден для коммерческих приложений обработки данных.

Метод моделирования данных приводит к успеху в силу следующих причин.

- Анализ данных используется как средство моделирования. Аналитики пытаются представить функциональность с помощью моделей данных. Модель данных устойчива по отношению к тем информационным требованиям, которые она описывает.
- Обрабатывающие компоненты — либо все подсистемы целиком, либо отдельные транзакции — взаимодействуют через совместно используемые данные, описываемые моделью данных. Выделение и стабилизация большинства важных системных интерфейсов на столь ранней стадии — это огромное преимущество.
- Возможен прямой переход от модели данных к предварительному проектному решению баз данных.

С этой точки зрения моделирование данных необходимо обобщить на случай моделирования объектов с разделением совместно используемых объектов и всех остальных. Объектно-ориентированный подход предоставляет более мощную среду моделирования, чем данные. Совместно используемые объекты образуют важные интерфейсы между системами. Кроме того, здесь все еще возможен прямой переход (хотя и несколько более длинный), связывающий модель совместно используемых объектов с проектным решением базы данных.

Совместно используемыми могут быть не только классы (по аналогии с определениями данных), но и экземпляры (по аналогии с данными). Совместное использование во многих системах накладывает определенные ограничения на диапазон методов, которые должны поддерживать совместно используемые объекты. Обработка, зависящая от конкретного приложения, должна быть исключена. В противном случае определение класса широко используемых объектов, например `Customers` (Потребители), должно непрерывно находиться в поле зрения, так как в каждом приложении к нему могут быть добавлены новые свойства. Создание отдельных подтипов для класса `Customers` (Потребитель) в каждом приложении невозможно, поскольку совместно используемые объекты являются самостоятельными объектами, которые не могут менять свой тип в каждом приложении. Зависящая от приложения функциональность моделируется вне рамок модели совместно используемых объектов на основании других объектов (например, объектов пользовательских интерфейсов), которые используют службы, предоставляемые совместно используемыми объектами.

Одно из преимуществ разработки систем на основе модели совместно используемых объектов состоит в том, что приложения могут быть относительно независимыми и программы могут независимо взаимодействовать с базой данных. Одно приложение может быть дополнено новыми или измененными объектами или же объекты могут быть удалены при минимальном воздействии на другие приложения при условии, что интерфейсы совместно используемых объектов остаются без изменений.

Немногие из хорошо известных методов объектно-ориентированного анализа и проектирования уделяют особое внимание разграничению совместно используемых и остальных объектов. Тем не менее это является необходимым условием для успеха метода объектно-ориентированного анализа в области коммерческих вычислений.

Диаграмма, представленная на рис. Б.47 с объяснениями, приведенными в табл. Б.2, описывает технологию моделирования совместно используемых объектов более подробно. Здесь важна высокая степень интеграции. Отдельные части имеют эквиваленты почти во всех структурных методах. Модель данных эквивалентна представлению модели объектов. Динамическое поведение объектов является иным полезным представлением модели объектов. Эти описания разработаны как представления, описывающие скорее объединенные в одно целое модели, а не отдельные модели, подлежащие последующему объединению. При такой интеграции моделирование на основе совместно используемых объектов отличается от структурных методов и даже в некоторой степени от метода OMT.

Моделирование графического пользовательского интерфейса рассматривается как специальный вид анализа. С целью определения задач и результатов используются контекстные диаграммы, а также аналоги диаграмм прецедентов. Далее применяются соответствующие принципы проектирования (например, CUA от фирмы IBM, разработанный в 1991 году). Проектирование окон отделено от проектирования пользовательских объектов. В результате получается трехуровневая модель, в которой окна запускают службы пользовательских объектов, находящихся на следующем уровне и, в свою очередь, использующих службы совместно используемых объектов.

Для решения проблем, связанных с реализацией объединения при создании интерфейсов реляционных баз данных, используется механизм, схожий с представлениями баз данных — "многие маложивущие объекты реализуют один долгоживущий объект".

Метод SEOO представляет собой гибридный метод, значимость которого обусловлена многоуровневой моделью совместно используемых объектов, а также тем, что основное внимание в нем уделяется эволюционному отходу от существующих технологий.

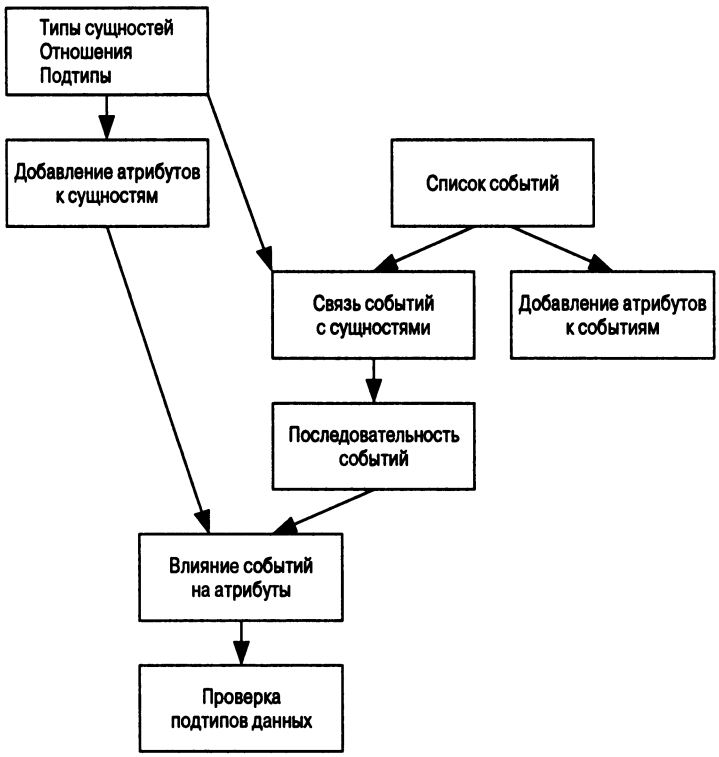


Рис. Б.47. Моделирование совместно используемых объектов

Таблица Б.2. Пояснения к рис. Б.47

Типы сущностей, отношения и подтипы весьма похожи на используемые в модели данных (сущность-связь), но интерпретируются как статическое представление модели совместно используемых объектов. Термин "сущность" относится к объектам и классам, указанным в модели совместно используемых объектов
Добавление атрибутов к сущностям. Атрибуты интерпретируются как запросы к сущностям. (Согласно [544], запросы представляют собой методы или "процедуры", которые возвращают значения, но не влияют на состояние.)
Список событий обеспечивает анализ динамики бизнес-области. Событие происходит в бизнес-области и приводит к изменению состояния сущности. (В терминологии Мейера (Meyer) событие называется "командой". В действительности событие является более сложным, поскольку одно событие может повлиять не на одну, а на несколько сущностей.)
Связь событий с сущностями. Перекрестная ссылка

Окончание табл. Б.2

Добавление атрибутов к событиям. С каждым событием может быть связано некоторое значение. Например, финансовая транзакция описывается суммой и датой
Последовательность событий. Описание возможного порядка событий. Используется сокращенный вариант системы обозначений из [359]. Нарушение этих ограничений считается ошибкой
Влияние событий на атрибуты определяет семантику событий (команд) посредством задания их влияния на атрибуты (запросы). Как правило, это делается неформально и зачастую выборочно, так как подавляющая часть результатов воздействия очевидна
Проверка подтипов данных выполняется с целью подтверждения, что выделенные подтипы соответствуют событиям и ограничениям на порядок следования (удовлетворяют правилам подстановки). Проверка подтипов подразумевает пересмотр иерархии в модели совместно используемых объектов. (Согласно правилу подстановки, клиент объекта, имеющего тип А, должен являться клиентом объектов, тип которых является подтипом типа А.)

Метод BON

В работе [585] описывается метод BON (Better Object Notation — улучшенное объектное представление). Он является одним из весьма немногочисленных методов, которые серьезно учитывают значимость бизнес-правил и инвариантов классов. Этот метод разрабатывался при финансировании в рамках программы European ESPRIT II (европейской стратегической программы исследований в области информационных технологий), и на него повлияли методы Буча, Коада-Йордона, Пэйдж-Джонса и Константина, Шлеер-Меллора, OMT, OOSD CRC и работа над объектно-ориентированным методом Z (см. работу [240]), что делает метод BON поистине гибридным. Основное внимание в нем уделяется неразрывности анализа и проектирования, масштабируемости, реверсивности, трассируемости, а также статическим и динамическим моделям и управлению компонентами. В рамках метода BON выполняются следующие виды деятельности.

1. *Определение границ системы.* Сюда входит также идентификация внешних событий.
2. *Идентификация классов-кандидатов.* Классы в методе BON определяются при помощи атрибутов, операций, ограничений и отношений с другими классами. Какие-либо специальные способы идентификации объектов отсутствуют.
3. *Группировка классов в кластеры.* Кластеры представляют собой подсистемы или пакеты в стиле языка UML, не имеющие четкой семантики, но явно не рассматриваемые как объекты. Они применяются для группировки связанных наборов классов.
4. *Определение классов-кандидатов на основе вопросов, команд и ограничений.* Аналитик задает вопрос: "Какие данные можно затребовать?" Ответы становятся атрибутами или "функциями". Команды являются операциями, а ограничения представляют собой либо утверждения, либо инварианты классов. Инварианты классов можно считать правилами, а ограничения описывают знания, поддерживаемые объектами.

Пытаясь определить иерархию классификации, аналитики проясняют поведение объектов с помощью вопросов. Ассоциации записываются довольно необычным образом — в виде текста.

5. *Определение поведения каждого класса на основе событий, протоколов сообщений между объектами и диаграмм создания объектов.* События могут быть внешними или внутренними. Внутренние события обычно зависят от времени и приводят к определению протоколов сообщений и утверждений, связанных с глобальным управлением системой. Система обозначений, предназначенная для описания классов, предусматривает использование специальных символов для указания входящих и исходящих потоков данных. Пред- и постусловия, а также инварианты классов записываются при помощи системы обозначений формальной логики. Пример инварианта класса, констатирующий, что клиентам корпорации не предоставляются скидки в выходные дни, выражается следующим образом:

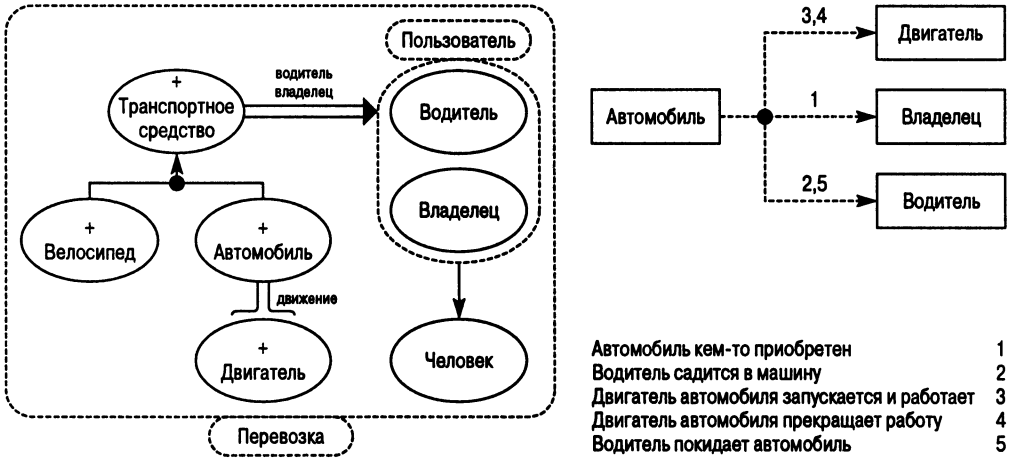
```
client.is_corporate ⇒ ∀l∈documents • ¬l.discount.weekend_rate
```

На структурированном английском языке это можно записать гораздо проще. Диаграммы создания объектов показывают, какие классы создают экземпляры других классов.

6. *Определение свойств, инвариантов и контрактных отношений классов.* Свойства классов могут соответствовать внутренним событиям. Статические и динамические отношения изображаются на отдельных диаграммах с использованием необычной и весьма своеобразной системы обозначений.
7. *Уточнение описаний классов.* Здесь выясняются возможности повторного использования.
8. *Разработка структур классификации.*
9. *Завершение создания архитектуры и ее редактирование.*

Индексирование классов позволяет выбирать их из библиотеки по стандартному шаблону заголовка. Метод BON обладает некоторыми сильными сторонами, которых нет у многих других методов, например в нем уделяется особое внимание управлению компонентами и правилам. Тем не менее автору не нравится его система обозначений. Однако он вызовет большой интерес у разработчиков, пишущих на языке Eiffel, поскольку этот метод непосредственно поддерживает конструкции языка, оставаясь в то же время независимым от какого-либо конкретного языка.

По своей сути метод BON близок к методу SOMA, описываемому в следующем разделе. Для него существует CASE-средство, основанное на совместимом с переносимой универсальной инструментальной средой PCTE репозитарии и получившее название EiffelCase. Имеется также средство создания диаграмм и система управления компонентами. Некоторые из обозначений метода BON представлены на рис. Б.48 и Б.49.



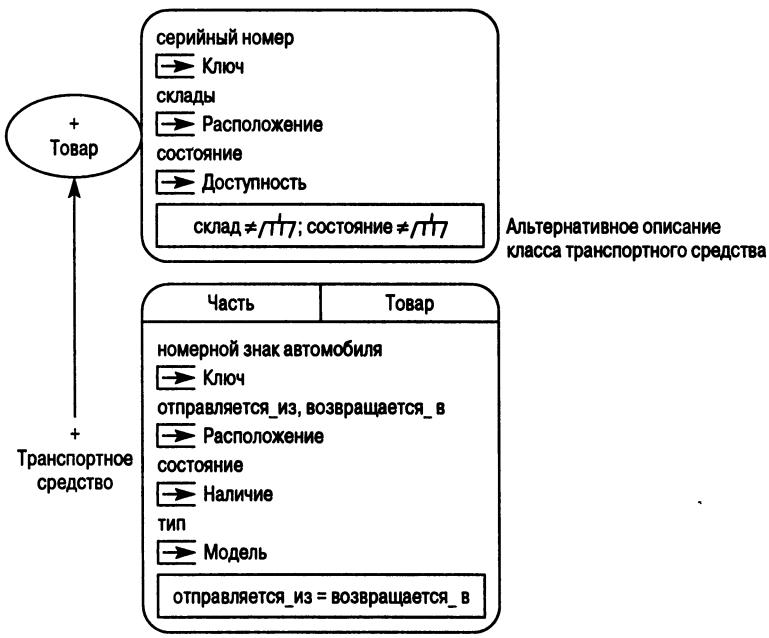


Рис. Б.48. Фрагменты системы обозначений метода BON

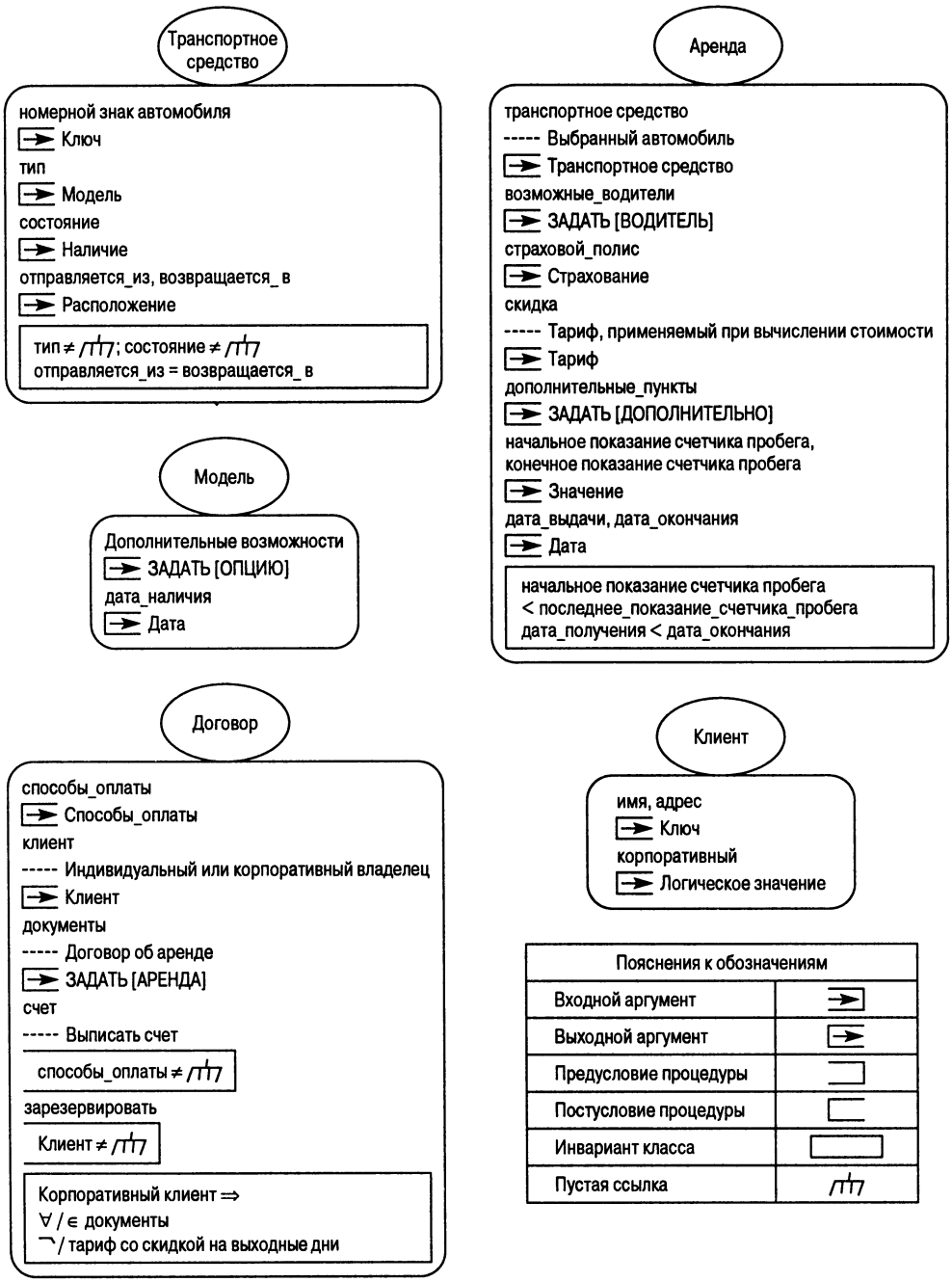


Рис. Б.49. Описания классов для системы проката автомобилей, выполненные в системе обозначений метода BON

В рамках статической модели, представленной на рис. Б.48 (слева вверху), кластеры показаны в виде пунктирных именованных прямоугольников с закругленными углами. Классы показаны в виде эллипсов, внутри которых содержатся их имена. Они могут содержать дополнительные сведения.

- Имена абстрактных классов начинаются со звездочки.
- Неабстрактные производные классы помечены знаком плюс.
- Имена многократно используемых классов подчеркнуты.

Отношения наследования обозначаются одинарной линией со стрелкой, причем стрелка указывает направление от потомка к родителю. Отношения клиент/сервер обозначаются двойной линией, которая в случае ассоциации заканчивается стрелкой, а в случае агрегации — открытой фигурной скобкой. Рядом с двойной линией могут быть указаны характеристики класса, участвующие в реализации отношения клиент/сервер. Отношения определяются между классами и могут быть распространены на кластеры.

В динамической модели объекты представлены в виде прямоугольников. Для обозначения множественных экземпляров можно использовать тень. Средства коммуникации показаны пунктирными линиями, которые снабжены последовательными номерами, относящимися к текстовым сценариям.

Очевидно, вся эта система обозначений может быть представлена при помощи UML.

Fusion

Метод Fusion (дословно — слияние) (см. работы [178, 179]) представляет собой гибридный метод, разработанный командой под руководством Дерека Колемана (Derek Coleman) в лаборатории компании Hewlett-Packard (Великобритания). В 1990 году компания Hewlett-Packard сделала обзор используемых ею методов. В результате был получен набор требований, предъявляемых к методам объектно-ориентированного анализа и проектирования. Основное требование касалось модели использования, обладающей простой системой обозначений. Ни один метод не удовлетворял абсолютно всем потребностям компании Hewlett-Packard. Метод OMT предоставляет технологию анализа, но лишь эвристику для проектирования.

При создании метода Fusion многие его идеи были позаимствованы из других методов. И он все еще продолжает разрабатываться. Основное влияние оказали следующие методики: система обозначений метода OMT, моделирование взаимодействия из метода проектирования на основе обязанностей RDD, идеи метода Booch91 относительно областей видимости и некоторые идеи, позаимствованные из школ формальных методов Z и VDM. Кроме того, в нем была реализована идея прецедентов, предложенная в методе Objectory. Модель процесса, описанная в соответствии с методом Fusion, представлена на рис. Б.50.

Динамическая модель метода OMT (а также динамические модели других методов, например метода Шлеер-Меллора) была признана бесполезной для применения на практике и поэтому не использовалась. Это подтверждает мнение автора об излишнем акцентировании внимания многих методов на моделях состояний. В функциональной модели не применяются диаграммы потоков данных, поскольку они являются излишне операционными. Вместо этого используются пред- и постусловия, хотя иногда они кажутся мне слишком сложными для понимания пользователями. Методы вводятся только на стадии проектирования.

Параллельность не поддерживается, и поэтому в методе даже нет условий инвариантности. Поскольку такие утверждения не связаны с классами, они не могут быть унаследованы.

Для некоторых специалистов-практиков, выполняющих объектно-ориентированный анализ, одна из непростых проблем состоит в определении момента прекращения анализа. Метод Fusion утверждает, что он может помочь в решении этой проблемы. Области видимости вводятся на стадии проектирования. Диаграммы взаимодействия определяют, каким объектам и какие методы нужны — и опять же на стадии проектирования. Для модели взаимодействий используются карточки CRC, однако соглашения не рассматриваются. Это можно сделать во время ролевых игр.

Рис. Б.50. Модель процесса в рамках метода Fusion

OBA

Метод OBA (Object Behaviour Analysis), основанный на анализе поведения объектов, описан в [672]. OBA представляет собой метод, который начинается с выявления сценариев в ходе переговоров или на основе документов (они похожи на прецеденты) и разработки контекстной модели. Затем идентифицируются участники и их обязанности и выделяются инициаторы. Видоизмененные карточки CRC используются для записи деталей объектов, а также выяснения структур классификации и использования других ассоциаций, хотя агрегация отдельно не выделяется. И наконец, для каждого объекта разрабатываются модели перехода из состояния в состояние.

Метод OBA в большей степени, чем подавляющее большинство всех остальных методов, охватывает требования, однако он не является чем-то особенно выдающимся в других областях. Подход к описанию требований, рассмотренный в общих чертах в главе 7, реализует некоторые из этих идей.

Syntropy

Метод Syntropy разработан Стивом Куком (Steve Cook), Джоном Дэниелсом (John Daniels) и их коллегами в компании Object Designers Limited [188]. Согласно утверждениям, этот метод не должен зависеть от какой-либо конкретной системы обозначений, однако

по умолчанию он использует популярную систему обозначений метода OMT. Этот метод основан на поведенческом подходе и рассматривает, главным образом, понятие поведенческой конкретизации (классификации). Некоторые механизмы метода Syntropy были позаимствованы из метода Booch91. Основная ценность метода Syntropy заключается в том, что в нем использованы способы формального описания типов и диаграммы состояний, выделены классы и типы, а также сделано различие между "реальными" моделями (т.е. моделями реального мира) и моделями анализа. Метод Syntropy оказал сильное влияние на метод Catalysis. Согласно некоторым высказываниям, метод Catalysis сделал для UML то же, что и метод Syntropy для OMT.

MOSES

Своим названием этот метод обязан Хендерсону-Селлерсу (Henderson-Sellers), который описал его в книге [372] и других публикациях. На его систему обозначений оказала влияние единая система представления объектов (Uniform Object Notation) Пэйджа-Джонса (Page-Jones), но она является более простой. Этот метод является гибридным; он предназначен для объектно-ориентированного анализа и основан на ранее опубликованной работе, принадлежащей нескольким авторам. В ней подчеркивается необходимость использования существующих структурных методов везде, где существует такая возможность. Я полагаю, это свидетельствует о современной тенденции, направленной на применение гибридных методов. Как и все остальные методы, это метод оказывает незначительную помощь при идентификации объектов. Система обозначений охватывает все важные структурные характеристики объектно-ориентированной системы. Правила не поддерживаются. Самым значительным новшеством явилась "фонтанная" модель жизненного цикла, где созданные элементы передаются вверх и "проливаются" вниз, а не закручиваются подобно воронкам спиральной модели. Представление классификации и других отношений на одной и той же диаграмме, как это делают другие методы, (заслуженно) обескураживает. Описание прецедентов в работе Хендерсона-Селлерса [372] немного смущает автора. Например, библиографический объект разбивается на объекты вида "сортировка" и "выход", смысл которых заставляет меня думать о них, как о методах, если я правильно понял. В методе предусмотрены рекомендации по поводу реализации моделей на различных объектно-ориентированных языках. Этот метод был расширен, уточнен и получил название MOSES, однако был вытеснен с рынка методом OPEN [334, 381].

Метод MOSES [378] поддерживает методологию объектно-ориентированной разработки программного обеспечения и основывается на расширении единой системы представления объектов, хотя допускает и другие системы обозначений. В нем предлагается использовать комплексную модель жизненного цикла и акцентируется внимание на неразрывности представления, рекомендациях по управлению проектом и возможности расширения. В этой работе применялась фонтанная модель.

Метод Текселя (Texel)

Рассматриваемый метод представляет собой метод объектно-ориентированного анализа и проектирования с очень глубокими корнями, уходящими в мир разработки на языке Ada, хотя утверждается, что он поддерживает разработки на языке C++ с учетом наследования. Описанная здесь версия была расширена для использования детальной модели процесса, похожей на каскадную модель, а также прецедентов, для описания которых применялись системы

обозначений из методов Booch93, OMT и UML [756]. Метод поддерживался компанией P.P. Texel and Co. штата Нью-Джерси и его создателем — Путнамом Текселем (Putnam Texel). Кроме того, компания обеспечивает обучение и может предоставить большое число примеров использования метода. Хотя почти все они относятся к приложениям из военной области и реального времени, утверждается, что приложения из области административных информационных систем поддерживаются с не меньшим успехом.

Действия, предпринимаемые в рамках оригинального метода Текселя, можно описать следующим образом. Сначала определяются типы объектов-кандидатов (которые называются классами объектов), атрибуты и т.д. Это можно рассматривать как описание системы (например, построение диаграмм потоков данных или словарей данных) на основе интервью с экспертом в данной области. Каждое существительное, глагол и т.д. на этой стадии записываются произвольным образом. Не дается никаких специальных рекомендаций относительно способов проведения интервью или идентификации объектов. К каждой фразе добавляются "ключи размещения", которые показывают, нужно ли считать данный элемент классом, атрибутом, дубликатом, потомком, процессом и т.д. В ходе этого процесса "список классов объектов-кандидатов" преобразуется в "список размещения". Следующим продуктом является список основных классов объектов BOCL, который представляет собой список классов, соответствующий этой стадии. И повсюду основное внимание уделяется трассируемости. На основании списка BOCL создается ER-диаграмма (Entity-Relationship), похожая на диаграмму информационной структуры Шлеер-Меллора (модель сущностей), показывающая ассоциации, но не агрегацию. С точки зрения семантики система обозначений ER-диаграммы является неполной, так как она показывает множественность и модальность, но не отображает уникальность или разбиение. Диаграмма уточняется посредством добавления атрибутов, агрегации, соотношений трассировки и границ системы. Отсутствуют специальные способы определения размещения класса или атрибута [567]. Конечные автоматы используются для последующего указания поведения, т.е. после создания обзорной диаграммы анализа классов объектов OOCAD (Overview Object Class Analysis Diagram) и производной от нее диаграммы анализа классов объектов OCAD (Object Class Analysis Diagram).

На рис. Б.51 и Б.52 показаны диаграммы OOCAD и OCAD для монитора состояния окружающей среды SEM (Sealed Environment Monitor). Затем создаются два документа: спецификация классов объектов OCSD (Object Class Specification) и спецификация отношений RSD (Relationship Specification), которые завершают статическую стадию анализа. Первая спецификация документирует классы, указанные на диаграмме OCAD, вторая описывает связи, показанные на этой диаграмме. Частично они представлены на рис. Б.53 и Б.54, где используется тот же пример монитора SEM.

Первая фаза метода Текселя схематически изображена на рис. Б.55. Вторая фаза анализа позволяет получить модели состояний для каждого класса. Для этого процесса характерен подход, похожий на методику Шлеер и Меллора [708]. Задача состоит в том, чтобы идентифицировать события и методы для каждого объекта. Конечно, это не играет большой роли для объектов, не имеющих значимого состояния и обычно присутствующих в бизнес-моделях. В этом заключается одна из причин, почему такие методы, как Fusion, избегают использования моделей состояний. Метод Текселя разрешает этот вопрос следующим образом: для "обиженных" объектов создается фиктивное состояние.

Одним из наилучших свойств метода является модель взаимодействия классов объектов OCCM (Object Class Communication Model), которая создается на этом этапе и позволяет анализировать потоки сообщений и событий. Таким образом, делается разграничение между внутренними событиями (которые идентифицируются в первую очередь) и внешними.

Сообщения показаны явно, однако эта система обозначений не может быть расширена до масштабов действительно сложных коммерческих систем. На рис. Б.56 представлена диаграмма OCCM для примера монитора SEM.

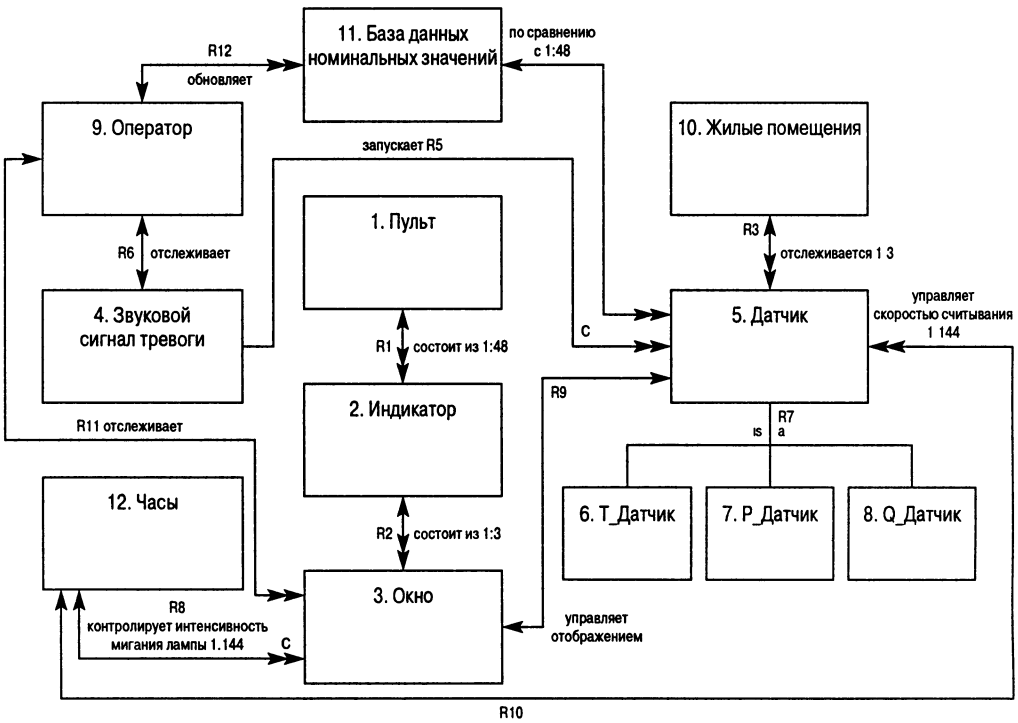


Рис. Б.51. Диаграмма OOCAD для монитора окружающей среды

В ходе заключительной стадии анализа создаются модели состояний для каждого класса, что иллюстрировано на рис. Б.57. На рис. Б.58 демонстрируется, насколько сложными могут стать эти модели состояний даже для таких простых вещей, как окна.

Множественное наследование запрещено. Сложные ограничения и правила не могут быть представлены непроцедурным образом, а утверждения не являются частью стадии проектирования.

Проектирование выполняется с использованием неформальной методики, похожей на метод Буча. Классы, которые станут объектами (пакетами в языке Ada) проекта, определяются и связываются при помощи общего основного алгоритма. Необходимость наследования сглаживается за счет предпочтения конкретных типов записей. Структурные схемы языка Ada используются для графического представления проекта, и затем генерируется код на языке описания программ (PDL).

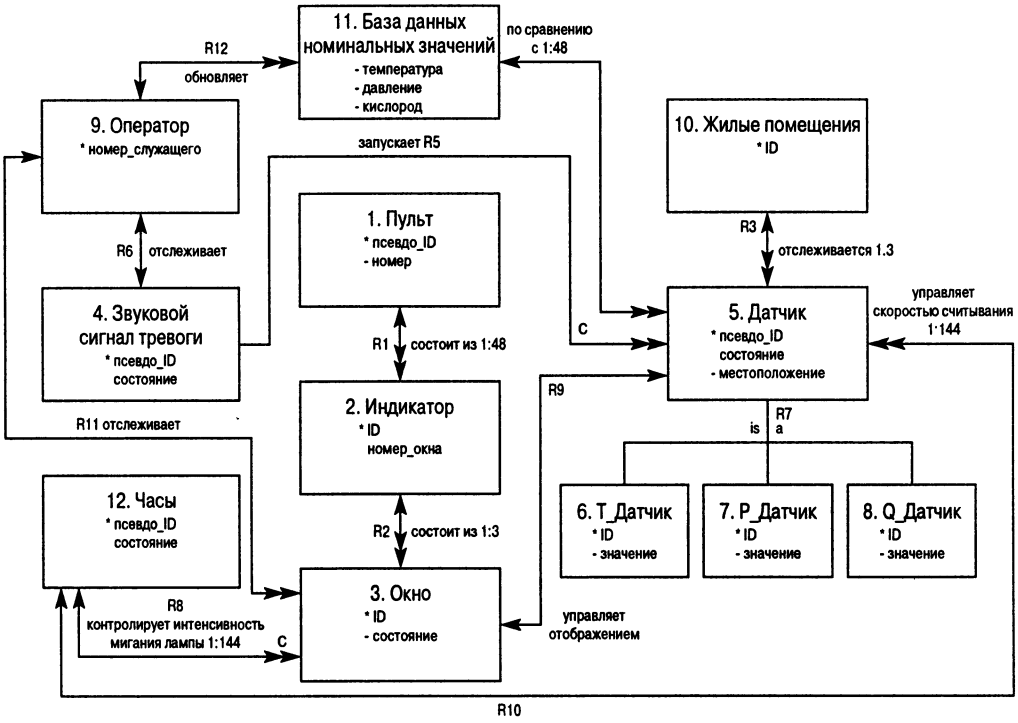


Рис. Б.52. Диаграмма OCAD

4. `Audible_Alarm` (Звуковой аварийный сигнал)

`Audible_Alarm (Pseudo_Id, State)`

Описание: Звуковой аварийный сигнал — это устройство, которое оповещает оператора, если значение одного из параметров окружающей среды отклоняется от нормальной величины на 3% или более.

С монитором HCC SEM связан только один экземпляр Audible_Alarm.

Audible_Alarm включается монитором HCC SEM и выключается оператором.

`4.1 Audio_Alarm.Pseudo_ID`

Описание...

Рис. Б.53. Фрагмент документа OCSD для описания звукового аварийного сигнала `Audible_Alarm`

На протяжении всего процесса огромное внимание уделяется созданию таблиц трассируемости, связывающих результаты анализа и проектирования. Автор не разделяет эту точку зрения, поскольку весь смысл объектно-ориентированного анализа и проектирования, несомненно, состоит в том, что трассируемость должна быть инкрементной и трассировочные метки должны воссоздаваться динамически, в ходе связанных между собой шагов.

```
R5.  Sensor (ЗАПУСКАЕТ) Audible_Alarm (Mc:1)
     Audible_Alarm (ЗАПУСКАЕТСЯ) Sensor
```

Величина отклонения реального значения, считываемого сенсором, от аналогичной величины, содержащейся в базе данных номинальных значений Nominal_DB определяет необходимость включения звукового сигнала.

Условие: Если значение, прочитанное сенсором, отличается от номинального значения в Nominal_DB на 3% или более, тогда и только тогда звучит аварийный звуковой сигнал.

```
R6.  Operators (ОТСЛЕЖИВАЮТ) Audible_Alarm (1:1)
     ...
```

Рис. Б.54. Фрагмент спецификации отношений RSD

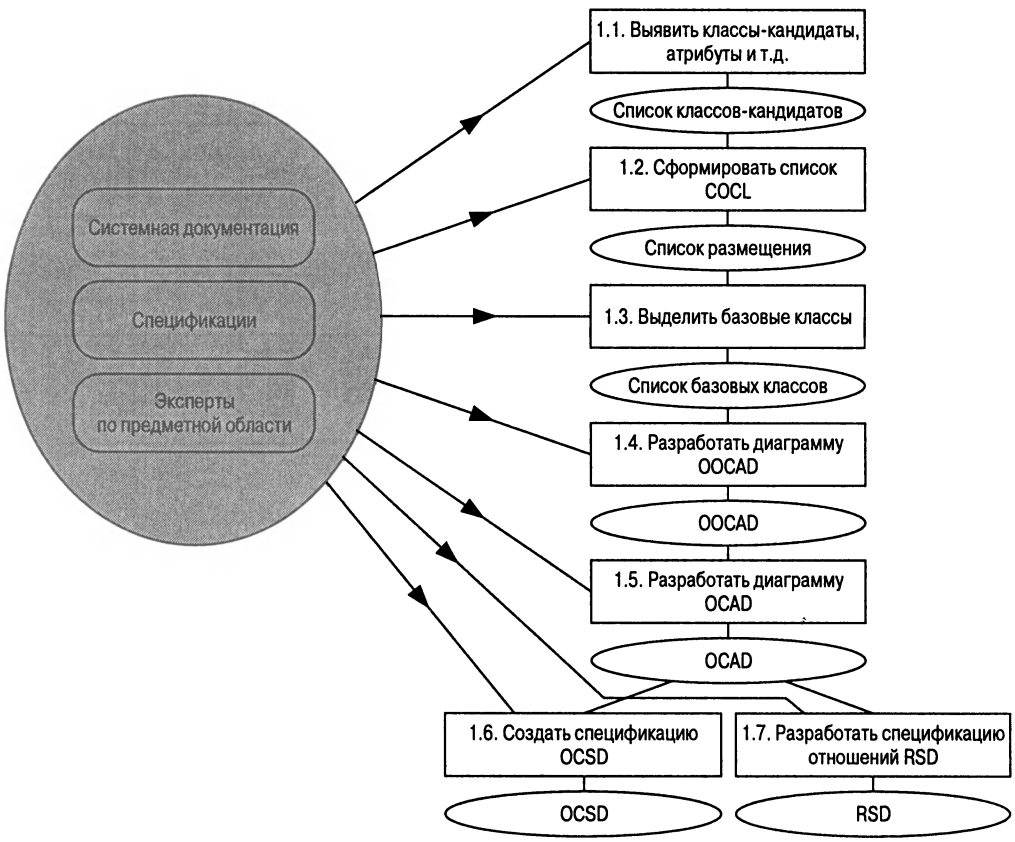


Рис. Б.55. Процесс анализа, описанный в соответствии с методом Текселя (фаза I)

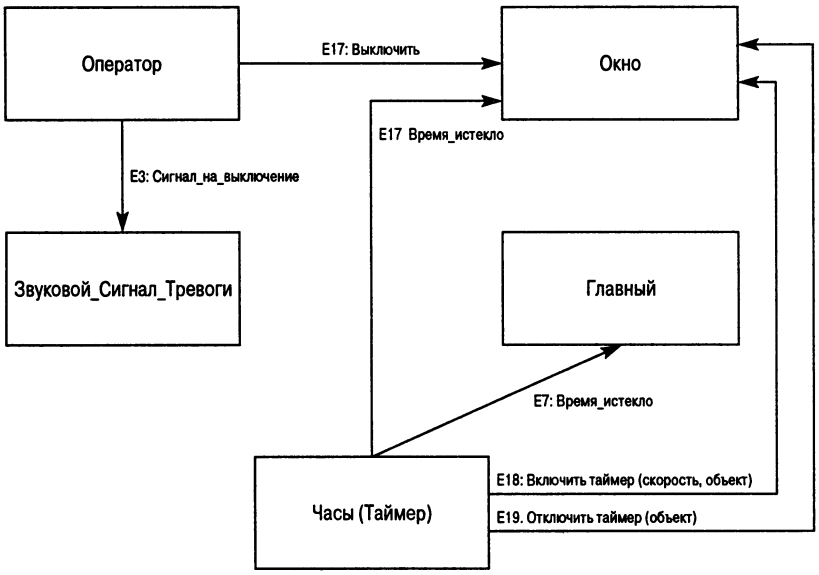


Рис. Б.56. Пример модели OCCM

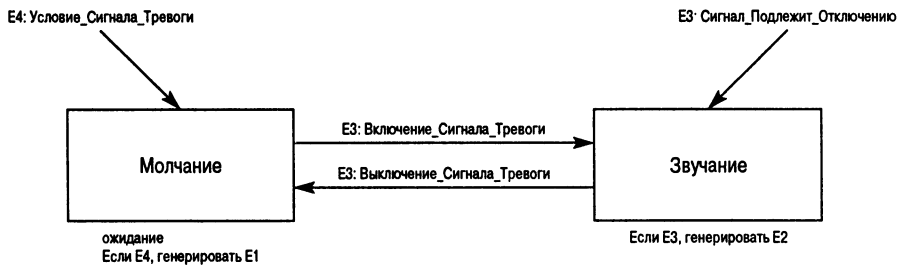


Рис. Б.57. Модель состояний для аварийного звукового сигнала, созданная в рамках метода Текселя

Метод Текселя включает отдельные ER-модели (сущность-связь), диаграммы переходов состояний и потоков сообщений. Его корни, связанные с проектированием систем реального времени, свидетельствуют о важности использования диаграмм переходов из состояния в состояние, где, как правило, не разрешается определять операции до получения модели состояний. Он тесно связан с методикой Шлеер-Меллора и приближается к методу Буча. Подобно все остальным методам, он не охватывает всех аспектов эталонной модели OMG (рабочей группы по развитию стандартов объектного программирования) и ему нечего предложить для идентификации объектов, управления компонентами, метрик или тестирования, за исключением таблиц требований/объектов, походящих на CRUD, которые снабжены комментариями, связанными с событиями. Понятия групп и представлений являются весьма ограниченными, исключение составляет понятие, схожее с понятием субъектов в методе Коада. Модель процесса скрыта, и в документации отсутствуют конкретные способы выполнения итераций, а также способы управления процессом создания прототипов.

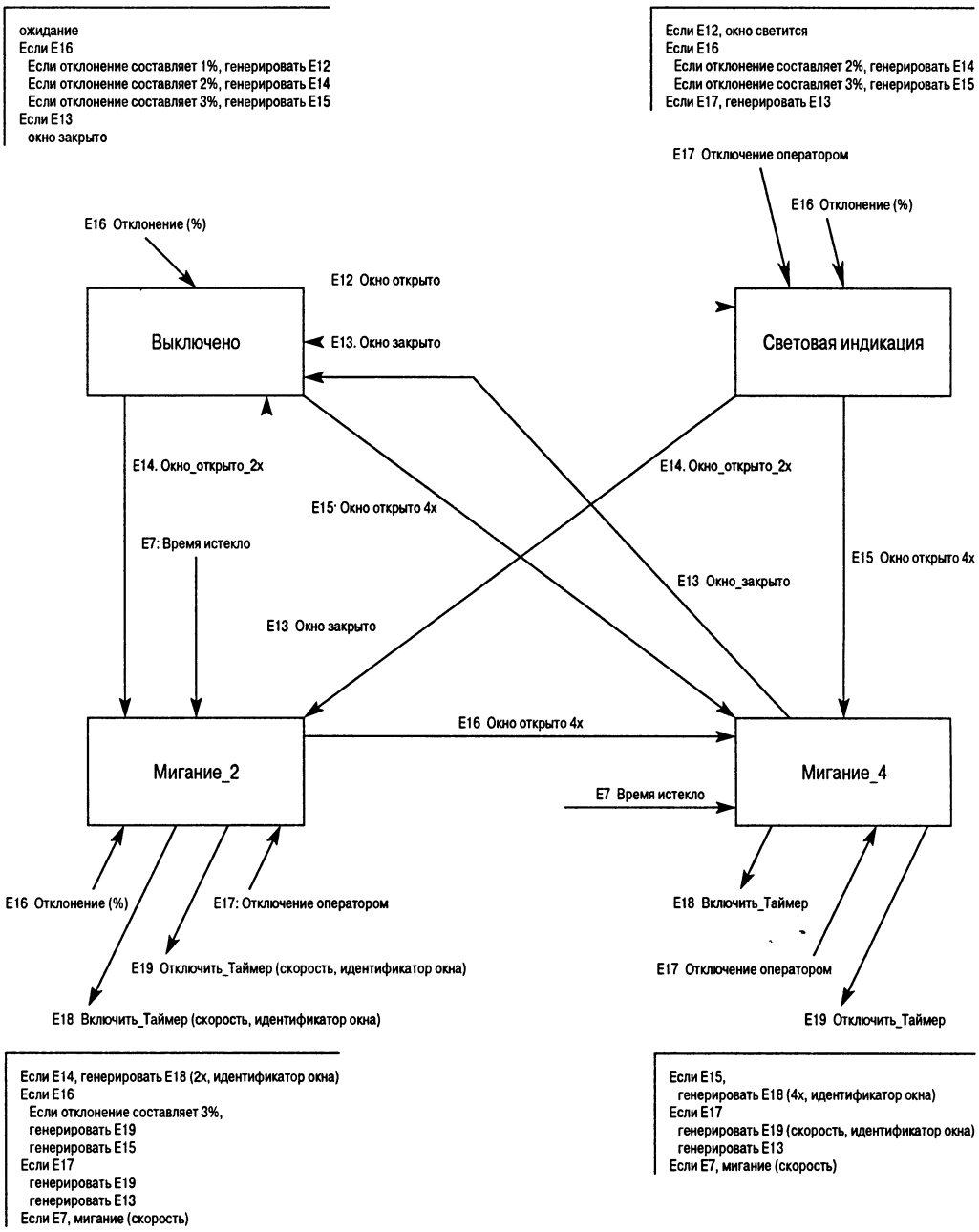


Рис. Б.58. Модель состояний в рамках метода Текселя

Основная проблема, связанная с этим методом, состоит в том, что трудно судить, насколько он может быть полезен в проектах, предусматривающих использование языка, не похожего на язык Ada, например языки Smalltalk, Eiffel или объектно-ориентированный язык программирования четвертого поколения[1]. Еще одна проблема касается бизнес-систем, и здесь можно предположить, что методы, создававшиеся для таких областей, как телекоммуникационные средства связи, приложения реального времени, оборона и тому подобное, менее пригодны, чем методы, которые своим происхождением обязаны традиционному семантическому моделированию данных.

Метод OORASS

Объектно-ориентированный метод ролевого анализа, синтеза и структурирования, OORASS (Object-Oriented Role Analysis, Synthesis and Structuring), был разработан компанией Taskon AS (Норвегия) [653, 654]. Он охватывает анализ и проектирование и необычен в том, что уделяет основное внимание ролям объектов, и затрагивает некоторые вопросы эволюционного жизненного цикла.

Анализ начинается с выяснения **областей интересов** (area of concern), которые представляют собой бизнес-функции высокого уровня и предназначены для формирования связанных подсистем. Затем моделируется каждая область, при этом используется сотрудничество агентов и объектов, которые могут исполнять различные **роли** (role). Роли могут быть определены посредством рассмотрения всех объектов, имеющих одно и то же положение в рамках структуры области интересов.

Роли интерпретируются таким же образом, как и роли агентов в методе Objectory, но эта идея распространяется и на внутренние свойства системы, не ограничиваясь только внешними агентами. Разные объекты в различных контекстах могут выполнять различные роли. Роли обладают требуемыми ресурсами (ссылки на серверы), компетенцией (то, что им известно), обязанностями и правами (операции). Роль X может "знать о роли Y", т.е. содержать ссылку, позволяющую отправлять сообщения. Как показано на рис. Б.59, такие связи клиент/сервер уточняются символами **портов** (port). Порт, обозначенный одним кружком, указывает на то, что роль X знает об экземплярах, выступающих в роли Y, число которых может составлять от нуля до единицы. Порт, обозначенный двойным кружком, свидетельствует о том, что Y знает об экземплярах в роли X, число которых может составлять от нуля до нескольких экземпляров. Отсутствие символа порта говорит о том, что роль на этом конце связи не знает о другой роли. Это позволяет вводить семантику множественности при моделировании объектов реального мира, представляя их на диаграммах использования, и может оказаться полезным при охвате требований высокого уровня таким же образом, как прецеденты в OOSE рассматриваются с учетом наследования.

С каждым символом порта можно связать набор операций (который получил название **соглашения** или **контракта**), определяющий, какую видимую роль он в состоянии предложить. Метод OORASS не зависит от системы обозначений, используемой для представления объектов. Понятие ролей соответствует интерфейсам языка UML.

---

[1] Программы на C++ можно разрабатывать в стиле языка Ada, но этот путь не является правильным. Как гласит народная мудрость, FORTRAN-программу можно написать на любом языке.

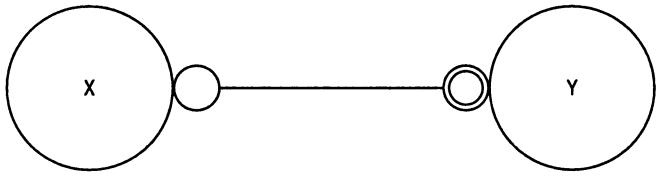


Рис. Б.59. Диаграмма ролей метода OORASS

Анализ рассматривается как нисходящий, но иерархический процесс, и в этом отношении можно провести аналогию с многоуровневым подходом метода SOMA, хотя модели ролей как таковые не подвергаются декомпозиции. Моделирование считается завершенным, когда аналитики имеют полное представление об интересующей области и модель устойчива. Если это не так или если область разлагается на составные части, специалисты, выполняющие анализ, должны переопределить область и повторить свои действия.

Типы, или типы объектов, синтезируются на основании ролей, что позволяет определять видимое поведение некоторых объектов. Во время **синтеза** (synthesis) ролей новые объекты создаются посредством наследования поведения от более простых объектов. Этот синтез повлиял на понятие инстанцирования шаблонов (т.е. создание экземпляра по шаблону — *примеч. пер.*) в методе Catalysis. Что касается терминологии, то метод OORASS определяет классы как реализацию типов и допускает существование различий между схемами типов и классов. Разрешено множественное наследование. В процессе структурирования применяется метамодель с целью задания поведения объектов на стадии выполнения, по мере того как они связываются друг с другом в процессе инстанцирования. Вызывает удивление тот факт, что подобная возможность отсутствует почти во всех других методах объектно-ориентированного анализа. Если подойти более формально, синтез начинается с области интересов и поиска событий и откликов, которые трансформируются в сообщения и операции. Идентифицируются подходящие объекты, хотя метод OORASS не предусматривает для этой цели никаких формальных способов; затем для идентификации и назначения операций и сотрудничества используются карточки CRC. Далее создаются диаграммы последовательности сообщений, а вслед за ними — диаграммы ролей. И наконец, очередность событий применяется для согласования контрактов с ограничениями и инвариантами.

Преимущество этого подхода состоит в том, что аналитики могут создавать простые модели ролей, представляющие взаимодействие объектов, а затем комбинировать их определенным образом с целью создания более крупных моделей. Утверждается, что при этом повышаются возможности многократного использования результатов анализа и проектирования и обнаруживается сходство этого процесса с философией прецедентов. Этот метод пытается решить проблемы, связанные с основными слабыми сторонами многих других методов, поэтому на стадии анализа уделяется особое внимание динамике систем во время выполнения.

Метод OORASS поддерживается CASE-средством OORAM, которое распространяется компанией Taskon. Этот метод предназначен для создания систем, в которых особое значение уделяется передаче сообщений и распределенности. Он поддерживает многие мощные и интересные идеи моделирования, которые, помимо всего прочего, должны быть полностью реализованы в основных методах.

Другие методы

Метод **ADM3** [271] представляет собой расширение более раннего, ориентированного на язык Ada метода ASTS, уделяющего основное внимание разработке систем реального времени. В нем предусмотрены системы обозначений для моделирования состояний, управления и временного моделирования, а также использованы идеи из области семантических сетей. Это комплексный метод, напоминающий в некоторой степени и метод Booch91, и метод OMT. Кроме того, метод ADM оказал влияние на разработку методов OPEM и OML.

Метод анализа фреймов и объектов, FOA (Frame-Object Analysis), описанный в [34], — это метод, берущий начало от концептуального моделирования, описанного в работах, посвященных усовершенствованным базам данных и искусственному интеллекту. В нем особое внимание уделяется использованию семантических сетей. Он охватывает довольно значительную часть жизненного цикла, от формулировки требований до тестирования, и необычен в своем выделении бизнес-правил. В этом методе фрейм является основным компонентом информационной системы, которая рассматривается как семантическая сеть, содержащая объекты и ассоциации. В действительности фреймы представляют собой подсистемы или слои и являются коллекциями объектов. Каждый объект обладает уникальностью, атрибутами и операциями. Диаграммы фреймов используются для отображения содержимого, классификации, ассоциаций, потоков данных, упорядочения и переходов состояний, и далеко неясно, каким образом можно избежать смешивания классификации и композиции или переходов из состояния в состояние с потоком данных. Сильной стороной метода FOA является то особое внимание, которое в нем уделяется бизнес-правилам и использованию ограничений и триггеров для их представления. Для этой цели предусмотрена даже поддержка небольшой дополнительной системы обозначений. Этот метод больше всего подходит для конструирования систем баз данных, где в качестве языка разработки должен использоваться язык C++, а для управления данными — усовершенствованная реляционная система.

CGI Yordon — это один из оригинальных, основанных на диаграммах потоков данных, методов Йордона (Yourdon). В нем предложено объектно-ориентированное расширение структурных методов Йордона (см. [520]) и предусмотрено соответствующее обучение. Чрезвычайно простой подход описан в работе [739]. Этот подход в значительной степени основан на использовании системы обозначений диаграмм потоков данных с целью описания внутреннего поведения объектов. Этот метод получает объекты из диаграмм потоков данных, логических диаграмм объектов и отношений или диаграмм переходов между состояниями, если такие традиционные модели были построены. Это удобно для тех специалистов, которые хорошо знакомы с философией Йордона, но непривычно почти для всех остальных объектно-ориентированных методов, которые занижают значимость потоков данных. Никакой другой метод не применяет потоки данных для описания самих объектов. Внешнее представление объекта описывается с использованием градиграмм (Gradygrams), отображающих индивидуальность объекта и его видимые операции. Наследование и ассоциации отображаются при помощи диаграмм в стиле Чена (Chen), и, кроме того, там, где это уместно, строятся диаграммы использования (зависимости объектов), причем их стиль соответствует стилю диаграмм старшинства метода HOOD. Внутренняя структура объектов представлена в виде обычных диаграмм потоков данных, где пунктирные линии показывают логику управления. Передача управления описывается при помощи диаграмм переходов состояний, таблиц решений или таблиц состояний. В противоположность почти всем хорошо известным методам объектно-ориентированного анализа, этот метод помогает различать объекты, имеющие сложные значимые состояния, но автор не убежден, что акцентирование внимания на потоках

данных полезно во всех других отношениях. Остальные методы стремятся использовать диаграммы потоков данных для контекстного моделирования.

В работе [616] разработан метод Synthesis, предназначенный для объектно-ориентированного анализа и проектирования. Позднее в работе [615] авторы ввели так называемую **единую систему обозначения объектов** (Uniform Object Notation) в контексте общего подхода к объектно-ориентированному анализу и проектированию, причем она имела много общего с обычной практикой проектирования программного обеспечения. Эта система обозначений похожа на некоторые другие системы представлений объектно-ориентированного анализа и проектирования, рассмотренные в этой книге; очевидное влияние на нее оказал метод Буча и структурное проектирование. Еще позднее Пэйдж-Джонс [614] обобщил классическое понятие Константина (Constantine), которое гласит, что в хорошем проекте минимизируется связывание и максимизируется зацепление посредством определения понятия связанности и трех различных видов инкапсуляции. Нулевой уровень инкапсуляции подразумевает, что строка кода инкапсулирует определенную абстракцию. Первый уровень — это инкапсуляция процедур внутри модулей, и второй уровень представляет инкапсуляцию в смысле объектно-ориентированного программирования. Два элемента системы являются связанными, если они имеют одну и ту же историю и будущее или, что более справедливо, если изменения одного неизбежно приводят к изменениям другого. Хороший проект должен исключать ненужное связывание, нарушающее границы инкапсуляции, и максимизировать его внутри этих границ. Для нулевого и первого уровней инкапсуляции можно ограничиться принципом связывания и зацепления. Как уже говорилось в главе 1, наследование подвергает риску многократное использование. Принцип связанности говорит о том, что при наследовании следует ограничиться видимыми характеристиками, т.е. должны существовать две отдельные иерархии наследования: для реализации и для интерфейса; эта мысль была высказана в главе 3. Кроме того, это воспрепятствовало бы использованию дружественных функций и классов в языке C++. Пэйдж-Джонс классифицирует некоторые виды связывания по имени, типу, величине, расположению, алгоритму, значению и полиморфизму. Связывание при полиморфизме представляет особый интерес для объектно-ориентированного проектирования и тесно связано с проблемами немонотонной логики. Например, если `FLY` (летать) — это операция класса `BIRD` (птица), а `PENGUIN` (пингвин) является подклассом класса `BIRD` (птица), тогда `FLY` (летать) иногда может быть неправильной операцией, а иногда правильной. При этом возникают проблемы, связанные с сопровождением: нужно ли изменять систему. Я полагаю, что во избежание подобной проблемы можно использовать и фаззификацию объектов. Этот вопрос рассматривается в приложении A. Принцип связывания имеет огромное значение и поэтому должен быть принят всеми проектировщиками, работающими в области объектно-ориентированных технологий.

В работе [71] предлагается достаточно всесторонняя объектно-ориентированная модель жизненного цикла, в чем-то напоминающая модель метода MOSES. Здесь также имеется влияние семантической сети в форме предложенных Берардом спецификаций классов и объектов. Они состоят из "конкретного и лаконичного описания" (определенного организационного резюме объекта), различных графических представлений, включая семантические сети и диаграммы переходов из состояния в состояние, списков требуемых и допустимых операций, констант и исключений. Система обозначений детализирована и, на мой взгляд, больше годится для проектировщика, нежели для аналитика. Правила не поддерживаются.

Метод **SOMA** (см. работы [312, 322, 327]) имеет собственную систему обозначений, основанную на анализе сущностей и связей ERA и методе Коада, но это теперь уже не представляет интереса. К его основным нововведениям можно отнести использование взаимосвязан-

ных наборов правил, инкапсулированных в классах, подход к проектированию требований и моделированию бизнес-процессов, акцентирующий внимание на задачах, и его моделирование жизненного цикла на основе контрактов. Все эти преимущества были изложены в данной монографии и поэтому не нуждаются в дальнейшем рассмотрении.

Метод **OPEN** (см. работы [272, 334, 381]) представляет собой комбинацию методов указанных авторов и некоторых других авторов из разных стран. Он тоже в достаточной мере был представлен в этой книге.

Метод основных объектов **Mainstream Objects** (см. работу [820]) представляет собой еще один аналог метода Fusion, причем его подход и система обозначений напоминают соответствующие характеристики метода SOMA. Он привнес в процесс много интересного.

Метод **OOSE** (Object-Oriented Software Engineering — объектно-ориентированное проектирование программного обеспечения) (см. работу [417]) представляет собой метод объектно-ориентированного анализа и проектирования, созданный на основе метода Objectory Якобсона (фабрики объектов). Метод Objectory был запатентован, и OOSE является упрощенной версией, которая вполне подходит для широкого использования. Метод Objectory выделяется среди других объектно-ориентированных подходов тем, что пытается охватить сразу весь жизненный цикл разработки программного обеспечения. Поначалу он возник из практики построения систем телефонных узлов в компании L.M. Ericsson. В нем применяются блочные проектные решения, и он является одним из старейших объектно-ориентированных методов. Другими своими корнями он обязан объектно-ориентированному программированию и моделированию данных. Многие из его идей, особенно прецеденты и способы тестирования, стали частью UML и/или RUP.

Существует также множество других методов. COOSD — это метод, разработанный компанией Aksit, расположенной в университете г. Твенте (Twente), Голландия. Он основан на композиции в стиле функциональной модели данных. К новаторским ключевым идеям можно отнести использование ролей и абстрактные типы средств сообщения. Следует упомянуть и метод ORCA (Object-Oriented Requirements Capture and Analysis). Объем книги не позволяет в полной мере охватить все идеи, но не вызывает сомнений тот факт, что гибридные методы будут появляться по мере развития этой области. К остальным методам, которые не могут быть подробно рассмотрены здесь, можно отнести методы ALEX-OBJ, MOOD, OSDL, OSMOSYS и SYS_P_O. Некоторые другие методы упомянуты в библиографических заметках, представленных ниже. К 1994 году существовало, вероятно, свыше 50 методов, которые можно было назвать более или менее полными. Очевидно, что такая ситуация не может считаться нормальной, особенно в силу того, что ни один из них не был завершенным и многие из них просто выражали взгляды специалистов по поводу того, что считать важным в разные моменты жизненного цикла. Пользователи должны были синтезировать законченный метод из фрагментов существующих опубликованных методов. В этом контексте появилась система обозначений UML, которая разрешила половину проблемы. Однако мы должны помнить, что метод — это не только система обозначений. Процесс синтеза хорошего метода все еще продолжается, и эта книга вносит свой вклад в эту дискуссию.

Б.3. Дополнительная литература

На развитие методов разработки очень большое влияние оказали ранние статьи Буча [93, 94], уже достаточно упоминавшиеся выше. Его более поздняя книга [98] представляла собой, пожалуй, самый глубокий и полный обзор первых объектно-ориентированных проектных

решений. Она содержала множество мнений и эвристик относительно того, из чего состоит хороший проект, как понимать объекты и применять объектную технологию. В работе [776] дается исчерпывающее описание метода OOSD и его сравнение с методами HOOD, GOOD и MOOD. Хорошим источником информации о методе HOOD является работа [666], которая содержит полное справочное руководство по версии 3.0 метода HOOD, а также замечательный учебный материал. Справочное руководство по методу HOOD версии 3.0 было опубликовано издательством Prentice Hall.

Метод OODLE и рекурсивная разработка описаны в работе [708]. Использованию карточек CRC посвящена хорошая работа [803], где приводятся также два полезных примера. Лоренц [495] описал систему обозначений и аналогичный метод, который охватывает больше проблем, хотя не настолько глубоко.

Переработанная версия метода Буча, появившаяся в 1993 году, содержала ряд значительных дополнений и исправлений, в том числе небольшое смещение акцентов в сторону анализа, во многом упрощенный и усовершенствованный набор обозначений связей (если сравнивать его с теми обозначениями, которые представлены на рис. Б.14), улучшенную совместимость с другими системами обозначений и методами (особенно с методами OMT и Objectory) и сокращенную версию системы обозначений, известную как "упрощенная нотация Буча". Интересно, что вторая редакция книги Буча почти полностью была посвящена языку C++, а не тем пяти языкам, которые он использовал для иллюстрации своих замыслов в первых работах.

Многие из рассматриваемых объектно-ориентированных методов представляли собой скорее взгляды, чем тщательно разработанные методы. Гибридные методы, базирующиеся на синтезе идей обычных методов, искусственного интеллекта и семантического моделирования, все еще появляются, и сейчас они неизменно используют UML.

В 1990-х годах появилось огромное число публикаций, посвященных объектно-ориентированным методам проектирования и анализа, однако лишь немногие прошли тщательную проверку временем. Работы [171–173, 225, 257, 417, 521, 673, 707, 708] можно отнести к исключениям. Метод BON описан в работе [585], а метод OBA — в работе [672]. Особенно не поддерживая, но и не критикуя, автор должен также упомянуть предложения [4, 15, 49, 152, 177, 218, 231, 309, 351, 406, 434, 446, 478-479, 538, 615, 774, 793]. В [220] описывается метод MERODE. Упомянутые в главе 7 методы Вирфса-Брока [803] и Лоренца [495] в качестве основного приема моделирования используют карточки CRC. Кроме того, в методе Лоренца применяется понятие прецедентов.

В обзоре [795] акцентируется внимание на разных аспектах, уже упоминавшихся здесь, однако не делается никаких предположений насчет расширения описанных методов. В [789] рассматривается несколько методов, в том числе и метод OSMOSYS. Поистине превосходный обзор и классификацию свыше 20 методов можно найти в работе [567]. Работа [378] представляет собой обновленную редакцию более ранней публикации [372], дающей обзор нескольких методов и исследующей метод MOSES.

В работе [71] вниманию читателей предлагаются многочисленные суждения и пояснения, а также обсуждается применение идей Константина (Constantine) по связыванию и зацеплению в объектно-ориентированных системах. Однако автор предпочитает более лаконичную трактовку, представленную в [614].

B

Краткое изложение системы обозначений UML

Краткий словарь, позволяющий, однако, создавать многочисленные комбинации, лучше тридцати тысяч слов, которые только мешают работе ума.

Поль Валери (Paul Valery)

Это приложение иллюстрирует почти все обозначения унифицированного языка моделирования UML (Unified Modeling Language). Я опустил только немногие редко используемые и нефункциональные элементы. Пояснения и примеры большей части символьных обозначений можно найти в главах 6—8. Мы представляем здесь версию 1.3 UML, и уверяю вас, что версия 1.4 не содержит реальных изменений в системе обозначений. Однако версия 2.0 может включать значительные исправления.

B.1. Обозначения для моделирования объектов

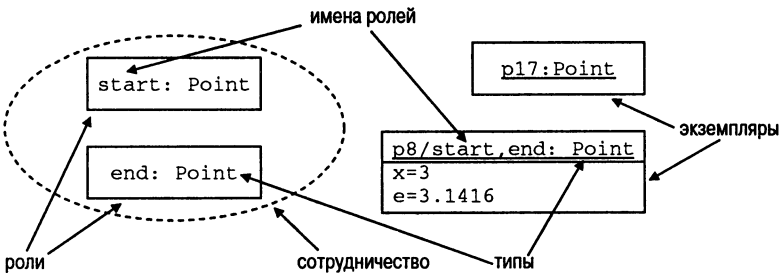


Рис. B.1. Экземпляры, роли и виды сотрудничества

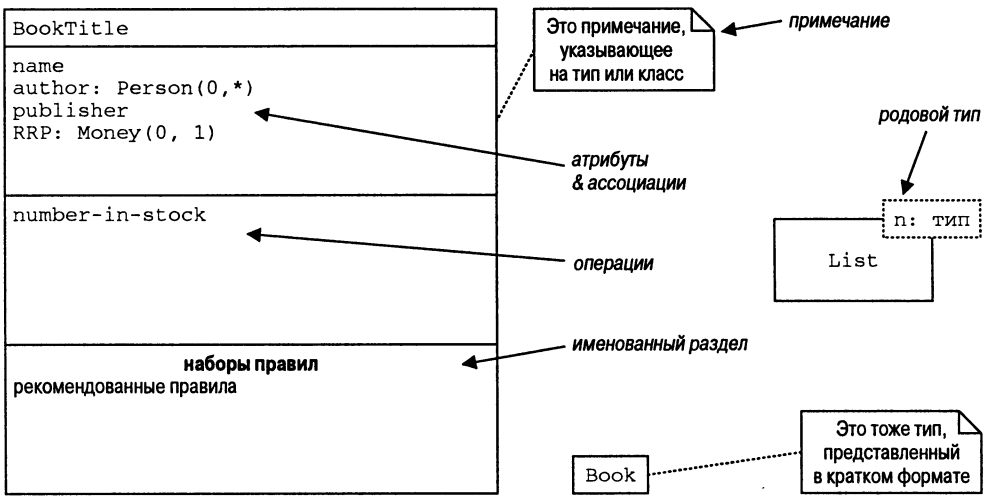


Рис. B.2. Классы и типы

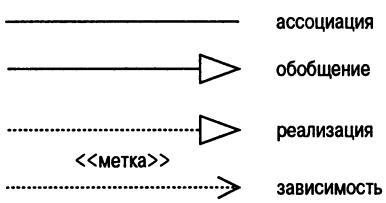


Рис. B.3. Символы для обозначения зависимостей

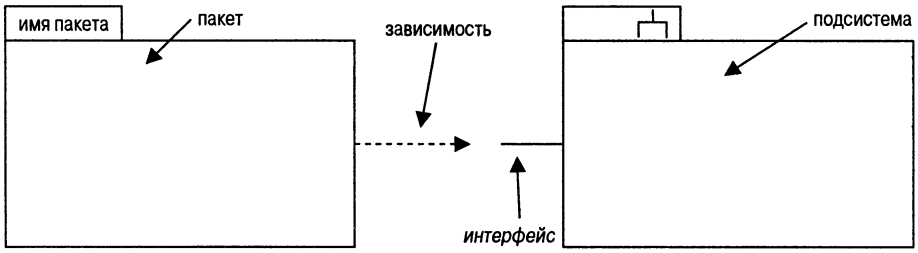


Рис. В.4. Пакеты и подсистемы

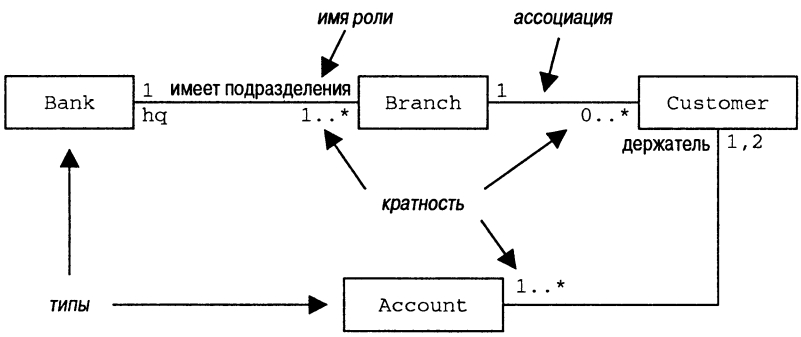


Рис. В.5. Ассоциации

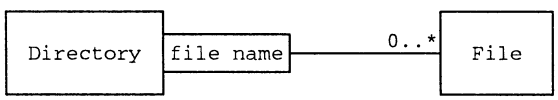


Рис. В.6. Спецификаторы

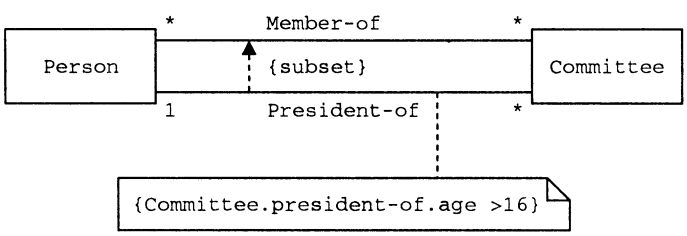


Рис. В.7. Неинкапсулированные ограничения. Некоторые ограничения задаются в UML заранее, например {ordered} (упорядоченный) и {or} (или) — см. рис. 6.7, а

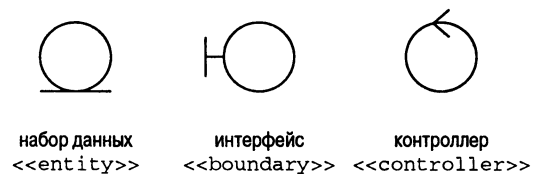


Рис. В.8. Визуальные стереотипы для классов

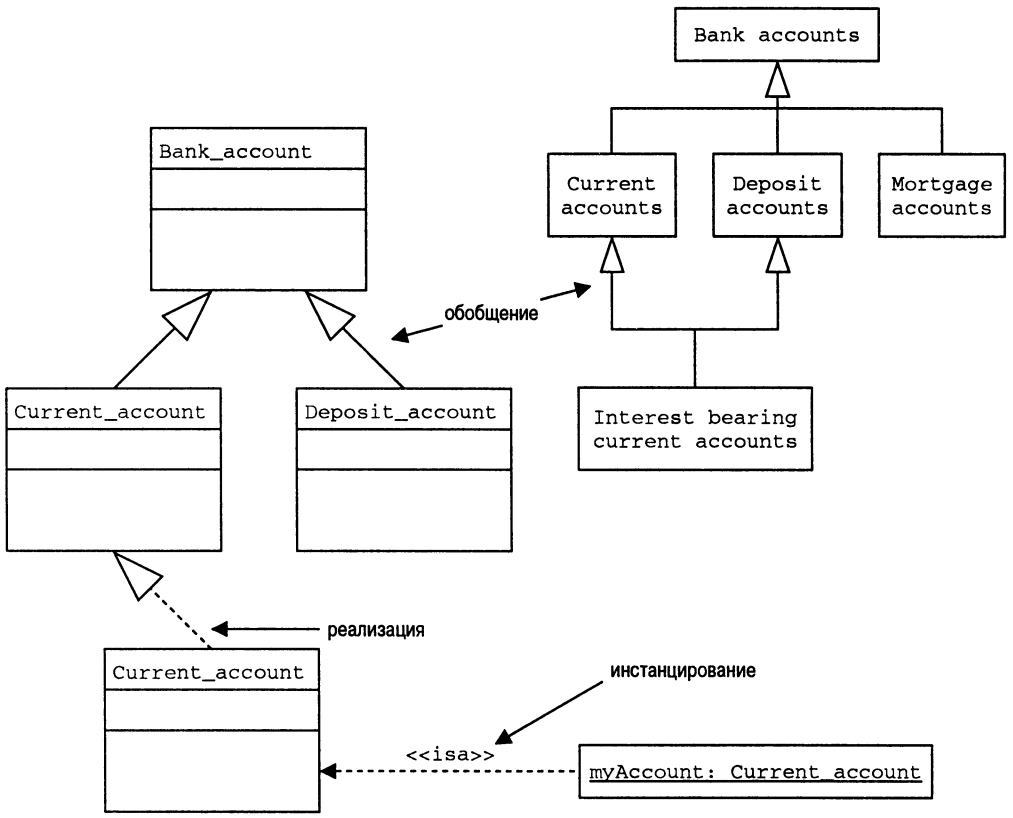


Рис. В.9. Наследование

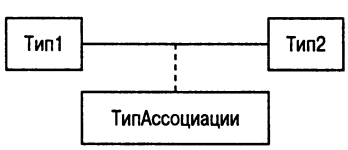


Рис. В.10. Типы ассоциаций

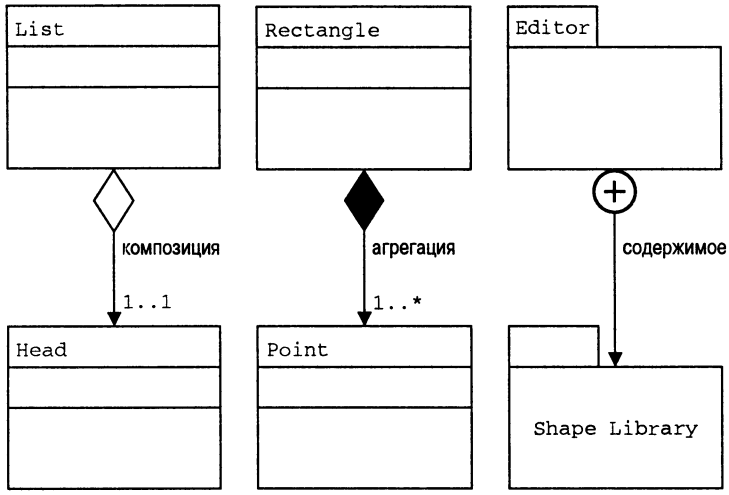


Рис. В.11. Агрегация

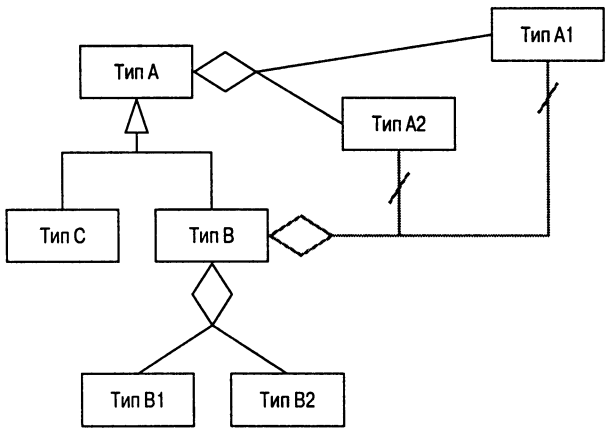


Рис. В.12. Производные зависимости

+ открытый – закрытый # защищенный

Рис. В.13. Области видимости для пакетов и классов

B.2. Обозначения для моделирования действий (прецедентов)

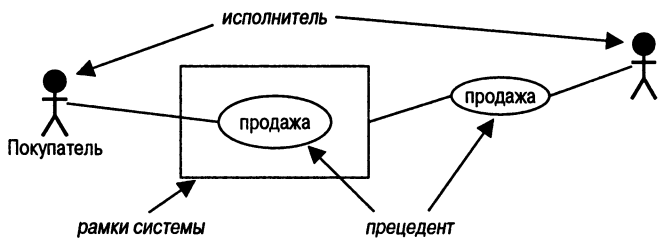


Рис. B.14. Исполнители и прецеденты. Зависимости, представленные на рис. B.3, могут также содержать значки прецедентов

B.3. Обозначения для диаграмм последовательностей и сотрудничества

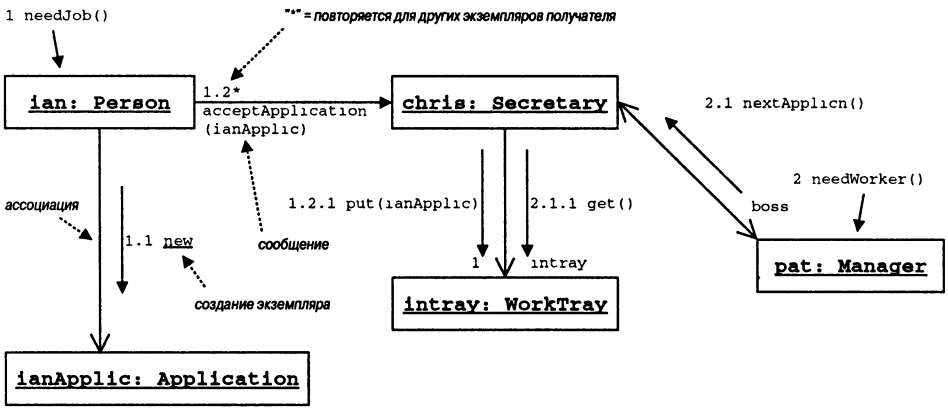


Рис. B.15. Система обозначений для диаграмм сотрудничества

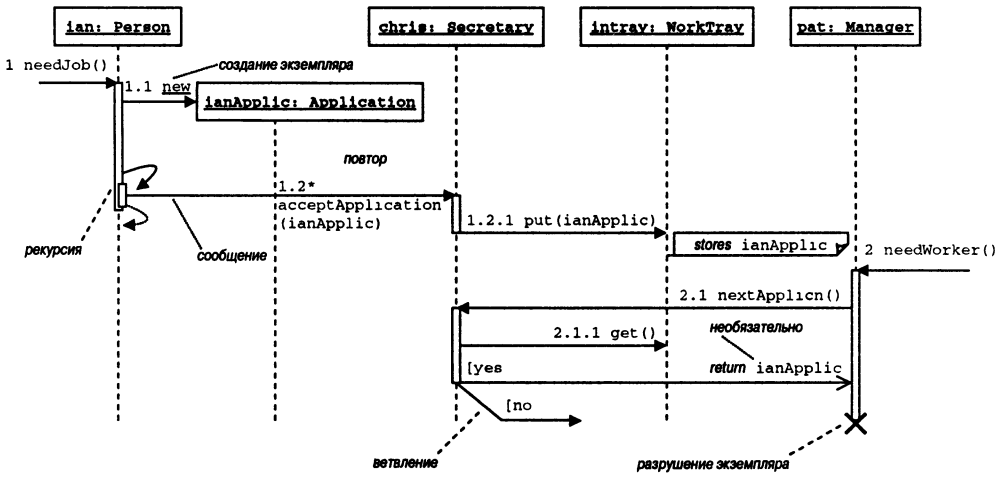


Рис. В.16. Система обозначений для диаграмм последовательностей

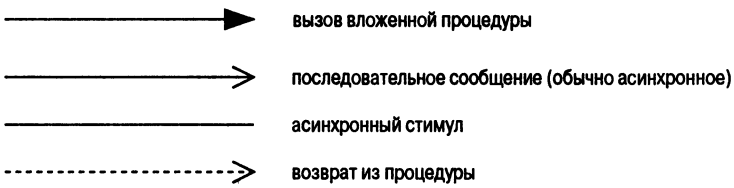


Рис. В.17. Типы сообщений

B.4. Обозначения для моделирования состояний

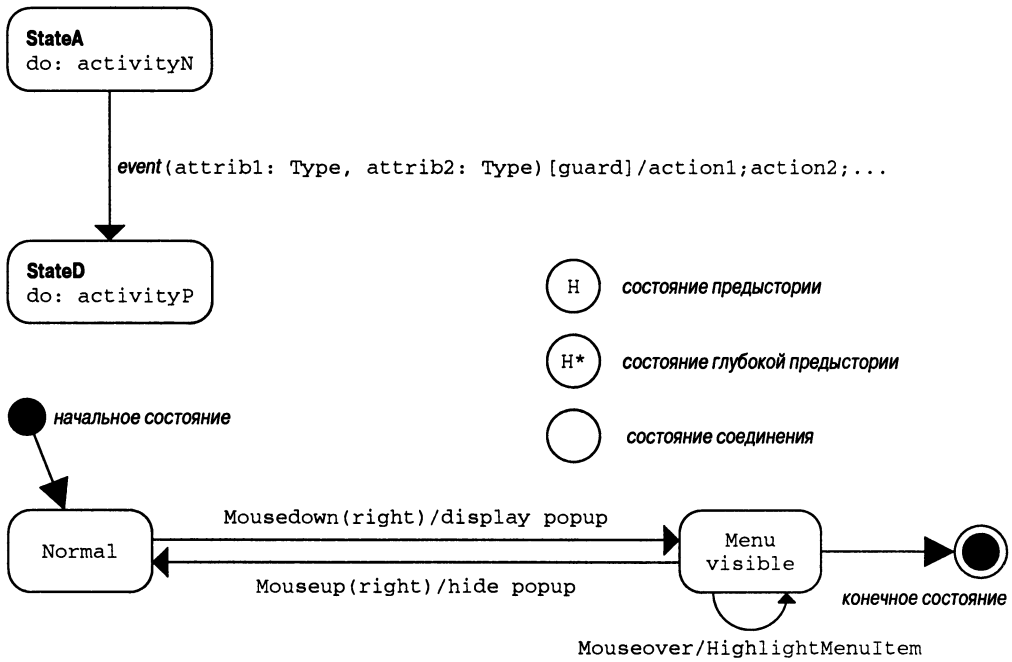


Рис. B.18. Система обозначений для диаграмм состояний

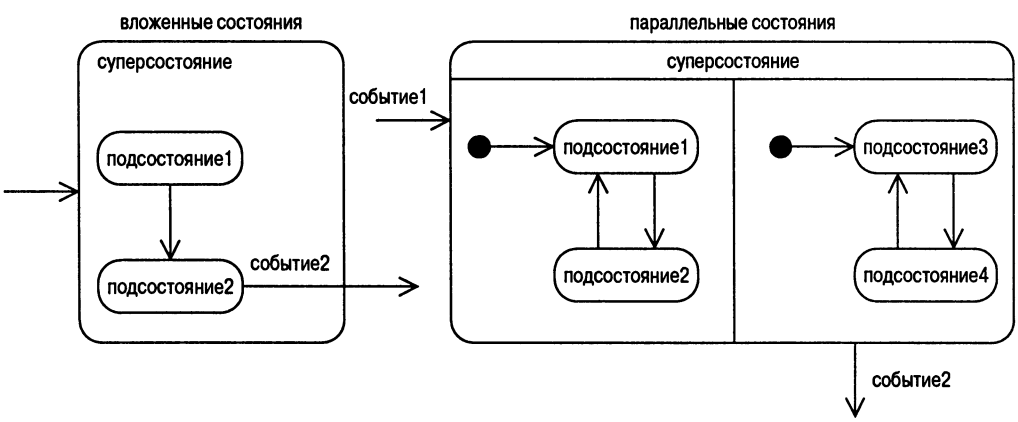


Рис. B.19. Вложенные и параллельные состояния

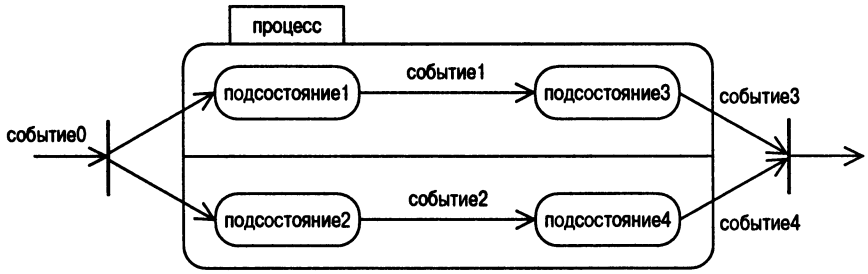


Рис. В.20. Бифуркация и синхронизация параллельных событий

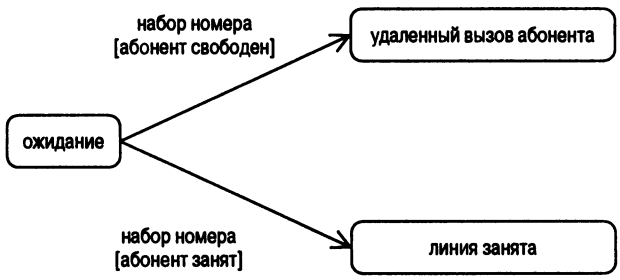


Рис. В.21. Защищенные транзакции

B.5. Обозначения для диаграмм действий

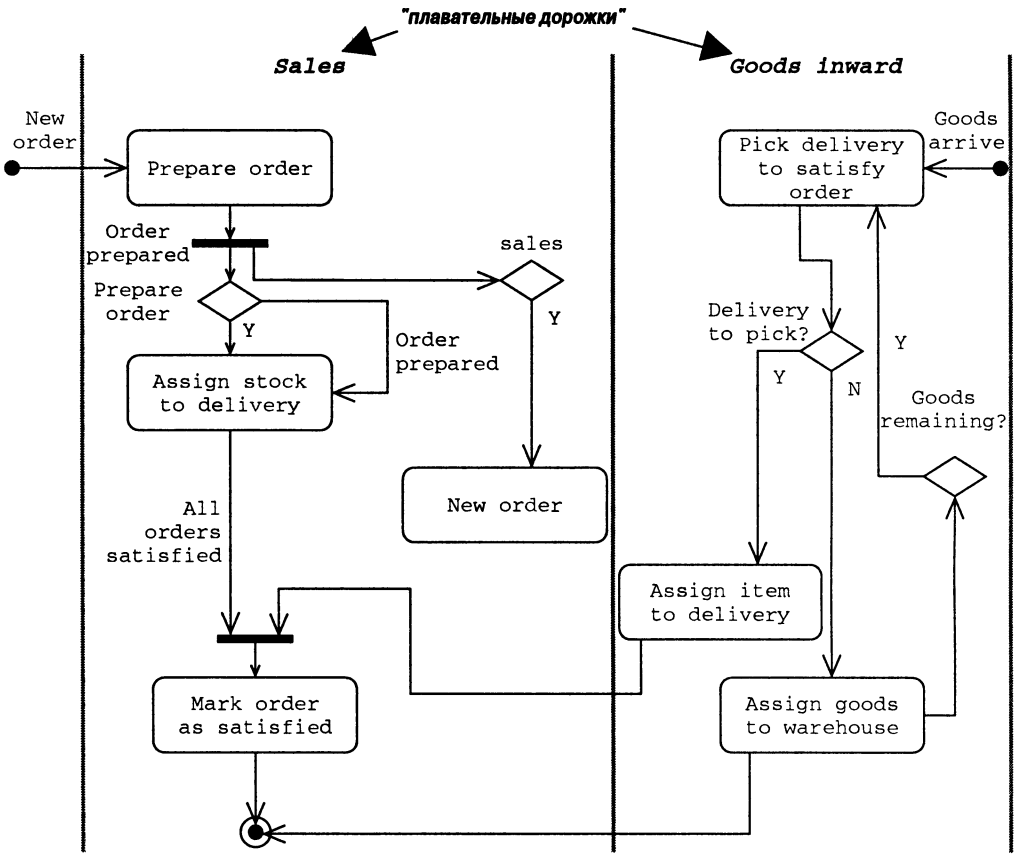


Рис. B.22. Диаграммы видов деятельности

В.6. Обозначения для моделей реализации и компонентов

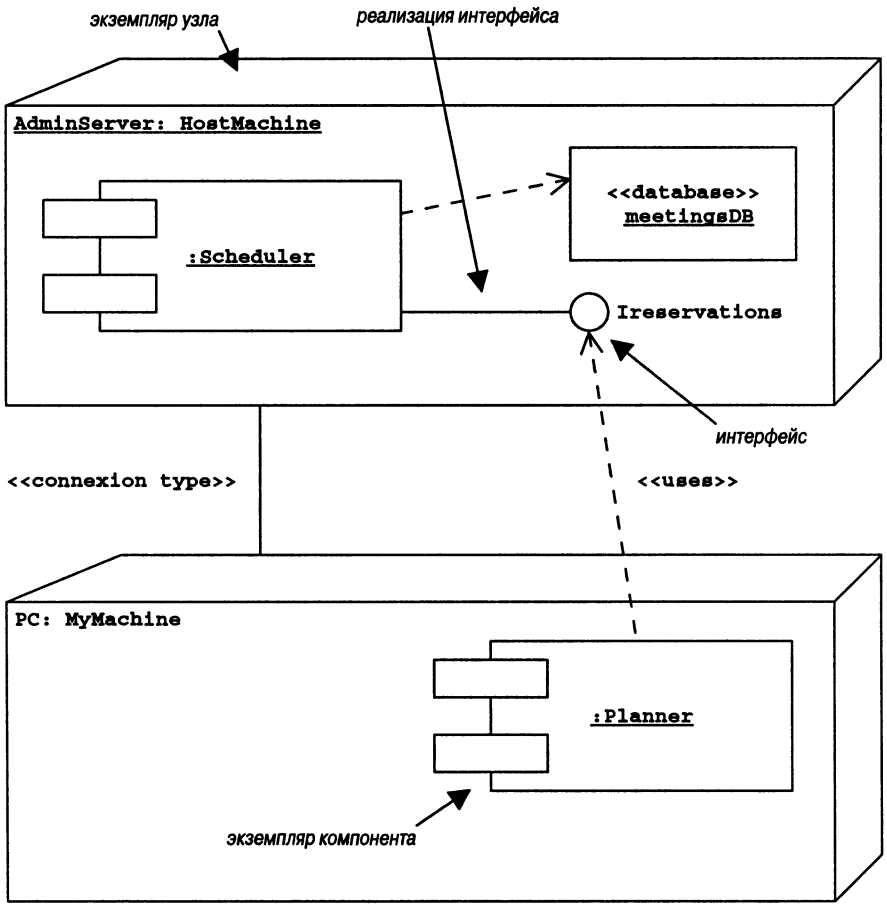


Рис. В.23. Узлы и интерфейсы

B.7. Виды сотрудничества и шаблоны

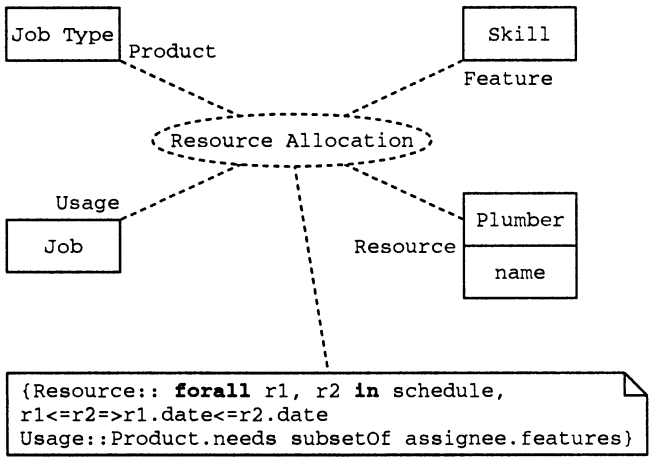


Рис. B.24. Виды сотрудничества. Эта же система обозначений применяется для представления шаблонов. Обозначения зависимостей, показанные на рис. B.3, могут использоваться для связи шаблонов с видами сотрудничества

B.8. Обозначения для систем реального времени

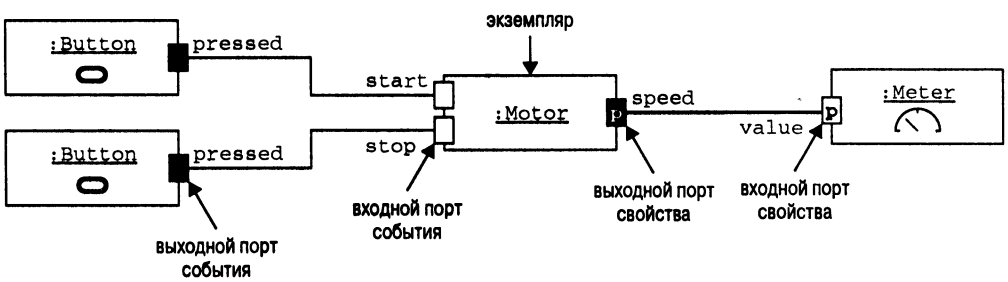


Рис. B.25. Порты и соединители

Словарь терминов

Выделенные курсивом термины включены в этот словарь. Расшифровка аббревиатур содержится в предметном указателе.

@ pre
Указывает значение атрибута перед выполнением действия. Является частью языка объектных ограничений OCL (Object Constraint Language).

ADT
см. *абстрактный тип данных* (abstract data type).

AKO (A kind of)
Отношение наследования между классами и их суперклассами: *обобщение* (generalization).

APO (A part of)
Отношение *композиции* (composition) между частями и целым.

BNF
Система обозначений, описывающая синтаксис языка программирования.

CRC (Class, Responsibility, Collaboration)
Карточка класса (class card), используемая как вспомогательное средство при обсуждении проектов.

IsA
Отношение классификации (наследования) между экземплярами и их классами, как, например, в следующем выражении: "Джейн Грэй — это ЧЕЛОВЕК".
См. *классификация* (classification).

OCL (Язык объектных ограничений)
Язык накладываемых на объекты ограничений (Object Constraint Language), определенный в UML. Язык, основанный на формальной логике.

t-норма (t-norm)
Обобщение различных видов операции "и" в нечеткой логике. Аналогичным образом t-конормы обобщают операции "или". Типичной нестандартной t-конормой является вероятностная сумма : $a+b+a*b$.

UML-RT
Расширение UML, предназначенное для систем реального времени.

XP
Экстремальное программирование (eXtreme Programming) — метод быстрой разработки приложений, уделяющий особое внимание автоматическому тестированию и коротким итерациям.

Абстрактный класс/тип (Abstract class/type)
Либо то же самое, что и *метакласс* (metaclass), либо (чаще всего) класс/тип, не имеющий экземпляров, но имеющий подклассы/подтипы. Абстрактный класс/тип противопоставляется *конкретному* (concrete) классу/типу. Иногда это класс, представляющий абстрактное понятие.

Абстрактный тип (Abstract type)
То же самое, что и *абстрактный класс* (abstract class), но без реализации. Тип, который должен задавать общее поведение для набора подтипов, или *абстрактный тип данных* (abstract data type).

Абстрактный тип данных (Abstract data type — ADT)
Похожая на класс абстракция, которая описывает набор объектов как инкапсулированную или скрытую структуру данных и операции, относящиеся к этой структуре.

Абстракция (Abstraction)
Определение существенных характеристик чего-то, не предусматривающее включения второстепенных или несущественных деталей.

Агент (Agent)
Объект, который выполняет некоторую операцию по требованию другого объекта и может, в свою очередь, воздействовать на другой объект, либо человек, организационная единица, программа или устройство, действующее в рамках предметной области или системы в кооперации с другими агентами; либо в языке Eiffel — конструкция, позволяющая осуществлять позднее связывание в стиле языка Lisp.

Агентная объектная модель (Agent Object Model — AOM)
Модель предметной области, созданная с использованием агентов и их *диалогов* (conversation).

Агрегация (Aggregation)
То же самое, что и *композиция* (composition). В UML — ассоциация на уровне экземпляров, которая является транзитивной и антисимметричной. Она задает семантику содержимого. Допускается множественная композиция.

Активный объект (Active object)
Объект, который может инициировать передачу сообщения, или объект, предоставляющий услуги для других объектов; либо объект, методы которого приводятся в действие при изменении его состояния; *демон* (demon).

Алгоритм (Algorithm)
Процедура, которая гарантированно завершается через определенный промежуток времени.

Анимация (Cartooning)
Представление агентов реального мира в *объектной модели бизнес-процессов* (Business Object Model) зачастую как интеллектуальных *агентов* (agent).

Антишаблон (Anti-pattern)
Стандартное, но неправильное или приносящее вред решение повторяющихся проблем; имеет имя, позволяющее специалистам свободно обсуждать его.
См. также *шаблон* (pattern).

Антропоморфизм (Anthropomorphism)
Моделирование объектов таким образом, как будто они проявляют человеческие качества.

Аплет (Applet)
Программа, которая работает в броузере и не сохраняется на машине клиента, а загружается в случае необходимости.

Аппликатор (Applicator)
Операция, в которой один из аргументов (ссылка на функцию) применяется к другим аргументам.

Ассоциация (Association)
Соответствие или пара соответствий между двумя типами/классами; указывает на то, что они могут обмениваться сообщениями.

Атомарная задача (Atomic task)
Задача или *прецедент* (use case), который при дальнейшем разбиении потребовал бы использования терминов, не относящихся к *онтологии предметной области* (domain ontology); обычно описывается одним предложением.

Атомарный (Atomic)
Неделимый на более мелкие элементы или структуры.

Атрибут (Attribute)
Статическое (или печатаемое) свойство объекта. Оно не может существовать независимо от объекта, хотя его значения — могут. Значениями атрибутов могут быть другие объекты, что делает их эквивалентными ассоциациям. См. *переменная экземпляра* (instance variable) и *ассоциация* (association).

Базовый класс (Base class)
Класс, от которого наследуется поведение других классов. *Суперкласс* (superclass).

Бизнес-объект (Business object)
Объект, упомянутый в анализе требований или спецификации, который имеет отношение к бизнес-процессу и описанию заданий (Грэм (Graham)); либо объект, который можно представить в виде значка, имеющего смысл для пользователей (Симс (Sims)); либо широко распространенный компонент программного обеспечения, который можно встретить в терминологии предметной области.

Бизнес-правило (Business rule)
Команда "если..., то...", касающаяся типов, атрибутов и операций *объектной модели бизнес-процессов* (Business Object Model).

Блокирующая отправка сообщения (Blocking send)
Сообщение, которое приостанавливает деятельность отправителя до тех пор, пока не будет получен ответ от сервера.

Брокер объектных запросов (Object request broker)
Программное обеспечение среднего уровня, поддерживающее нёзависимость от местоположения при обращениях к распределенным объектам; как правило, подчиняется стандартам CORBA (технологии построения распределенных объектных приложений фирмы IBM).

Броузер (Browser)
Инструментальное средство, помогающее программистам осуществлять визуальную навигацию по всем структурам классификации системы, перемещаясь по связям наследования; либо инструментальное средство для просмотра информации в Web.

Взаимная блокировка (Deadlock)
Условие, при котором обработка прекращается ввиду того, что каждая из двух операций ожидает завершения другой. Если не выполнить прерывание, это условие будет длиться бесконечно.

Вид деятельности (Activity)
Действие в рамках *проекта* (project), оперирующее определенными объектами.

Видимость (Visibility)
Способность одного объекта быть сервером для других.

Виртуальная машина (Virtual machine)
Программное обеспечение, имитирующее набор команд некоторых аппаратных средств на другом аппаратном обеспечении.

Виртуальные функции (Virtual functions)
Функции, не имеющие реализации. См. также *отложенные методы* (deferred method).

Влияние (Forces)
Влияние, подлежащее рассмотрению и балансировке в случае использования *шаблона* (pattern).

Внешний агент (External agent)
Агент, который находится за рамками текущей бизнес-области.

Возможность подстановки (Substitubility)
Принцип, гласящий, что все сообщения, отправляемые некоторому классу, должны быть понятны для всех его специализаций, поэтому такие специализации можно подставлять вместо имени класса.

Гарантия (Guarantee)
Утверждение, которое сервер должен поддерживать как истинное на протяжении всего времени выполнения операции (см. также *оператор переключения* (relay clause) и *условие инвариантности* (invariance condition)).

Граничные объекты (Boundary objects)
См. *интерфейсные объекты* (interface object)

Действие (Action)
Происходящее событие, приводящее к изменению состояния окружающего мира: задание, деятельность, прецедент и т.д.

Декларативная семантика (Declarative semantics)
То же, что и *функциональная семантика*.

Делегирование (Delegation)
В языках прототипирования на основе исполнителей так называется передача обязанностей по выполнению метода другому объекту. Представляет собой некоторую форму бесклассового наследования.

Демон (Demon или daemon)
Объект или процедура, активизируемая посредством изменения состояния. Процедура, запускаемая данными. Представляет собой то же самое, что и *trigger* (триггер).

Деонтическая логика (Deontic logic)
Логика гарантий и обязательств.

Диалектика (Dialectics)
Представление, в котором все постоянно видоизменяется посредством разрешения внутренних противоречий. Единство и борьба противоположностей.

Диалог (Conversation)
Целенаправленное взаимодействие между агентами, обладающее стандартной рекурсивной структурой; оно может включать до шести действий или прецедентов.

Динамическая классификация (Dynamic classification)
Позволяет экземплярам мигрировать из одного класса в другой (запрещено в языках объектно-ориентированного программирования).

Динамическое связывание (Dynamic binding)
Распределение памяти не на стадии компиляции, а на стадии выполнения. То же, что и *позднее связывание* (late binding). Противоположно по своему значению *статическому* (или *раннему*) (*static* или *early*) связыванию.

Дисциплина (Workflow)
Деятельность в рамках проекта (в RUP).

Дисциплина RUP (Rational Unified Process)
См. *дисциплина* (workflow).

Друг (Friend)
В языке C++ — метод объекта, имеющий привилегированный доступ к закрытой реализации другого объекта.

Зависимости (Dependencies)
Отношения между объектами, не являющиеся ассоциацией, композицией и наследованием.

Закрытые части, операции, методы, атрибуты, свойства (Private parts, operations, methods, attributes, features)
Такие части объекта, методы или атрибуты, которые недоступны для других объектов. Они доступны только для подклассов или экземпляров.

Защищенные части, операции, методы, атрибуты, характеристики (Protected parts, operations, methods, attributes, features)
Свойство называется защищенным, если доступ к нему имеют только подклассы.

Идентичность (Identity)
Объект (экземпляр, класс, тип или роль) обладает уникальной идентичностью на протяжении всей своей жизни.

Идентичность объекта (Object identity)
Правило, гласящее, что объекты обладают уникальной идентичностью в течение всего своего существования.

Идиома (Idiom)
Шаблон (pattern), обычно относящийся к определенному языку программирования.

Инвариант (Invariant)
См. *инвариант класса* (class invariant).

Инвариант класса (Class invariant)
Утверждение, касающееся атрибутов и/или операций класса и связывающее их.

Инкапсуляция (Encapsulation)
Область действия ссылки на объект. Объекты могут проверять или изменять собственное состояние, но их методы доступа запрещают другим объектам выполнять несоответствующие запросы и обновления. Это понятие тесно связано с понятием *сокрытия информации* (information hiding).

Инспектирование (Inspection)
Конструктивные встречи, в задачу которых входит выявление дефектов в получаемых объектах.

Инстанцирование (Instantiation)
Создание элемента данных, представляющего переменную или класс, путем задания значения.

Интеллектуальный агент (Intelligent agent)
Объект, обладающий способностью "мыслить": либо в реальном мире (человек, организация и т.д.), либо в системе (класс, содержащий *набор правил* (ruleset) и *метод вывода* (inference regime)).

Интерфейс (Interface)
Видимые методы объекта вместе с его инвариантами.

Исполнитель (Actor)
Объект, на который не действуют операции, но сам оказывающий влияние на другие объекты (Буч); либо объект, имеющий адрес и текущее состояние и способный связываться с другими объектами только посредством передачи сообщений. Исполнители могут изменять свое локальное состояние, создавать новых исполнителей и отправлять сообщения. В системах на основе исполнителей наследование заменяется делегированием (Хьюитт (Hewitt)); либо пользователь системы, играющий некоторую роль по отношению к этой системе (Якобсон).

Исполнитель (Worker)
Роль, которую принимает на себя член основной команды, работающей над проектом в рамках унифицированного процесса RUP (Rational Unified Process) компании Rational Software, который представляет собой методологию разработки, создания и распространения программного обеспечения.

Итератор (Iterator)
Метод, позволяющий просматривать любую часть объекта.

Каркас (Framework)
См. *каркас модели* (model framework) и *каркас приложения* (application framework).

Каркас модели (Model framework)
Связанная группа пакетов, абстрактные символы шаблонов которой можно заменять конкретной информацией. Базовая структура, как правило, представляет многократно используемый шаблон.

Каркас приложения (Application framework)
Коллекция объектов, предназначенных для решения проблем определенного класса. Широко распространенный пример — каркас приложения для разработки графического пользовательского интерфейса.

Карточка класса (Class card)
Представление объекта на бумаге, включающее стандартную схему, используемую для ролевых игр, критического анализа и проверки правильности объектных моделей. См. также *карточки CRC*.

Квалификатор (Qualifier)
Термин, который квалифицирует другой термин.

Квантификация (Quantification)
В логике — задание пределов, определяющих количество объектов, для которых применимо выражение. Классическая логика содержит два квантора: *существования* (записывается как ∃) и *всеобщности* (записывается как ∀).

Класс (Class)
Абстракция набора экземпляров, определяющая общие статические и поведенческие характеристики экземпляров, включая открытую и закрытую природу состояния и поведения. Класс представляет собой шаблон, на основании которого создаются экземпляры объектов.

Класс с отложенной реализацией (Deferred class)
Класс, содержащий какую-либо характеристику, представленную только в виде описания: ее реализация должна быть выполнена в некотором подклассе. В интерфейсе языка Java все характеристики являются отложенными. Язык Eiffel демонстрирует большую гибкость, позволяя объявлять отложенными отдельные характеристики.

Класс-контейнер (Container class)
Класс, который задает структуру данных, предназначенную для хранения экземпляров других типов, например списки, множества, наборы, очереди и стеки.

Клиент (Client)
Объект, использующий службы другого объекта — сервера, т.е. клиенты могут отправлять сообщения серверам.

Компонент (Component)
Определенный надлежащим образом объект, рассматриваемый вместе с атрибутами, операциями, инвариантами и видами сотрудничества; либо двоичная независимая единица реализации, накопления и развертывания, обладающая заданным интерфейсом и только явными контекстными зависимостями; либо единица развертывания, которая играет некоторую роль в наборе компонентов.

Конкретный класс (Concrete class)
Класс, который может иметь экземпляры.

Кэширование (Cashing)
Сохранение копии данных отдельно от действующей базы данных и обеспечение синхронизации между ними.

Логический вывод (Inference)
Процесс получения новых данных на основе существующих при помощи набора *правил* (rule) или *инвариантов* (invariant).

Манипулятор (Manipulator)
Значение, указанное в выражении и вызывающее побочные эффекты. Обычно реализуется через ссылки на другие методы и часто используется в качестве аргумента оператора *аппликатора* (applicator).

Метакласс (Metaclass)
В языках типа Smalltalk — это класс, экземпляры которого тоже являются классами, в противоположность *абстрактному классу* (abstract class), не имеющему экземпляров, а имеющему только подклассы. В некоторых случаях метакласс используется в более широком смысле как синоним абстрактного класса. См. также *суперкласс* (superclass).

Метод (Method)
Процедура или функция, которая может выполняться объектом. Строго говоря, реализация операции в некоторых языках, используемая в значении, близком к понятию *операции* (operation).

Метод класса/операция класса (Class method/class operation)
Метод (операция), который наследуется всеми подклассами и задает поведение набора экземпляров, а не отдельных экземпляров. Значения переменных класса могут наследоваться и перекрываться в подклассах.

Методы преобразования (Translational methods)
Методы, которые по мере продолжения разработки преобразуют объектные модели в новые формы.

Методы/операции экземпляра (Instance methods/operations)
Метод или операция, которые в противоположность *методу* или *операции класса* (class method/operation) воздействуют на отдельные экземпляры, а не на весь класс/тип в целом.

Множественная абстракция (Set abstraction)
Рассмотрение конкретных экземпляров как принадлежащих классу, заданному при помощи абстрактных свойств и предикатов.

Множественная классификация (Multiple classification)
Подход, в рамках которого экземпляр может принадлежать не одному, а нескольким классам (запрещен в языках объектно-ориентированного программирования).

Множественное наследование (Multiple inheritance)
Наследование, при котором некоторый более специальный объект может наследовать свои свойства от нескольких общих классов, в результате чего могут возникнуть противоречия.

Модальная логика (Modal logic)
Логика возможностей, которая дополняет обычную логику первого порядка двумя новыми операторами — L (означающим "возможно, что") и N (означающим "необходимо, чтобы"), а также новыми правилами, позволяющими манипулировать этими операторами.

Модель "классной доски" (Blackboard system)
Система, в которой объекты регистрируют заинтересованность в информации с помощью объекта "классной доски", управляющего планом и ходом решения проблемы.

Мономорфизм (Monomorphism)
Понятие, противоположное полиморфизму, согласно которому сообщение может обладать только одной интерпретацией в масштабах всей системы.

Мощные типы (Power types)
Типы, экземпляры которых тоже сами по себе являются типами.

Набор ассоциаций (Association set)
Именованный набор *ассоциаций* (associations) между прецедентами, эквивалентный диаграмме последовательности операций, предусматривающей параллелизм.

Набор ассоциаций задачи (Task association set)
Именованный набор ассоциаций между прецедентами или действиями; эквивалентен диаграмме последовательностей, содержащей информацию о параллелизме.

Набор компонентов (Component kit)
Набор компонентов вместе с заданными протоколами, предназначенными для их объединения.

Набор правил (Ruleset)
Совокупность команд "если..., то..." или *инвариантов* (invariant), рассматриваемая вместе с именованным режимом логического вывода. Правила считаются независимыми, и их порядок не имеет значения. В этом смысле языки, основанные на правилах, не являются процедурными.

Нагрузочная способность по входу/выходу (Fan-in и fan-out)
Нагрузочная способность по входу некоторого объекта — это количество клиентов, от которых объекту передается управление. Его нагрузочная способность по выходу — это число серверов, которые он использует.

Наследование (Inheritance)
Отношение между классами, благодаря которому один класс наследует часть описания другого, более общего класса, а экземпляры наследуют все свойства и методы тех классов, которым они принадлежат.

Наследование интерфейсов (Interface inheritance)
Наследование классом или интерфейсом описания свойств без их реализации.

Наследование класса (Class inheritance)
Наследование классом свойств и их реализации от суперкласса (суперклассов); в противоположность *наследованию интерфейсов* (interface inheritance).

Наследование, выполняемое без участия классов (Classless inheritance)
См. *delegation* (делегирование).

Настройка (Marshalling)
Преобразование параметров метода в формат, пригодный для передачи данных через брокер объектных запросов.

Независимость от местоположения (Locational transparency)
Возможность использовать объект, не имея при этом никаких сведений о его местоположении в распределенной системе.

Немонотонная логика (Nonmonotonic logic)
Формальная логика некоторого вида, теоремы которой могут быть отменены (бездоказательно) в свете новых фактов. Обычные логики являются *монотонными* (monotonic) в том смысле, что добавление новых фактов всегда приводит к увеличению числа доказываемых теорем.

Неограниченная универсальность (Unconstrained genericity)
См. *универсальность* (genericity).

Несоответствие импеданса (Impedance mismatch)
Напрасные усилия программиста, которые заключаются в выполнении преобразования между объектами и таблицами баз данных.

Нечеткий квантор (Fuzzy quantifier)
Термин, который неточно квантифицирует выражение, например, *несколько*, *немного* и *большая часть*.

Нечеткий набор правил (Fuzzy ruleset)
Набор правил, терминология которого предусматривает использование нечетких переменных. Выполняется при помощи нечеткого алгоритма прямого построения цепочки.

Нечеткий объект (Fuzzy object)
Объект, который может содержать или наследовать значения атрибутов, являющиеся нечеткими множествами, и который может наследовать любой атрибут, имеющий показатель достоверности, частично моделируя наследование.

Нечеткое множество (Fuzzy set)
Обобщение понятия множества, допускающее частичное членство. Множество высоких мужчин — это нечеткое множество.

Нечеткое наследование (Fuzzy inheritance)
Наследование атрибутов при наличии показателя достоверности. В случае множественного наследования они комбинируются в соответствии с правилом логического вывода.

Обобщение (Generalization)
Противоположно по своему значению *специализации* (specialization). Перемещение "вверх" по *структуре классификации* (classification structure). Более общий класс по отношению к некоторому другому классу, который унаследован от него.

Оболочка (Wrapper)
Разбиение диаграммы на несколько частей, трактуемых как композитные объекты, взаимодействующие с другими слоями.

Объект (Object)
Сущность, которая обладает уникальной идентичностью на протяжении всего времени жизни. Представляет собой либо тип, либо один из его экземпляров. Класс или экземпляр класса. Заметим, что в других книгах термин *объект* (object) используется только для обозначения экземпляров. Мы в своей книге придерживаемся философии языка Smalltalk, согласно которой "объектом является все".

Объект приложения (Application object)
Объект, который, скорее, имеет отношение к определенному приложению и не может быть многократно использован в рамках предметной области.

Объектная модель бизнес-процессов (Business Object Model)
Модель, выполненная при помощи UML и охватывающая все типы и действия, задействованные в описании бизнес-процессов; возможно, в форме исполняемой спецификации.

Объектно-ориентированный (Object-oriented)
Объектно-ориентированные системы являются *основанными на объектах* (object-based), основанными на классах, поддерживают наследование между классами и суперклассами и позволяют объектам отправлять сообщения самим себе.

Объектно-реляционные базы данных (Object-relational databases)
Реляционные базы данных, обладающие внешним интерфейсом, который придает им свойства объектно-ориентированных баз данных.

Объект-тип (Object-type)
Это стандартный термин рабочей группы по развитию стандартов объектного программирования OMG. Он подразумевает тип сущности с *операциями* (operations), инкапсулирующий структуру данных. Обычно обозначает *класс* (class), хотя, строго говоря, класс представляет собой реализацию объекта-типа. Иногда кратко упоминается как *объект* (object).

Объект-фабрика (Factory object)
Экземпляр, который отвечает за создание и удаление других экземпляров классов.

Объекты интерфейса (Interface objects)
Объекты, связанные с интерфейсом системы. Их называют также *пограничными объектами* (boundary object).

Объекты предметной области (Domain object)
Объекты, которые имеют значение не только для одного приложения и могут многократно использоваться.

Объекты-контроллеры (Controller objects)
Объекты, которые являются центрами логики системы, они вызывают другие объекты и управляют ими. Таких объектов следует избегать.

Обязанность (Responsibility)
То, что объект может знать (атрибут) или уметь выполнять (операция).

Ограничения кардинальности (Cardinality constraints)
Инварианты (invariant), которые задают количество объектов, на которые может указывать ассоциация; например, значения атрибута `DateOfBirth` (Дата рождения) могут принадлежать диапазону от нуля до единицы.

Ограниченная общность (Constrained genericity)
См. *genericity* (универсальность).

Онтология (Ontology)
Наука или учение о бытии. Кроме того, концептуальная модель, обосновывающая область знаний.

Онтология предметной области (Domain ontology)
См. *онтология* (ontology).

Оператор переключения (Relay clause)
Предусловие, которое должно оставаться истинным в течение всего времени выполнения операции, к которой оно относится. Спецификация не устанавливает, какой результат должен быть получен пользователем в случае нарушения этого условия.
См. также *гарантия* (guarantee) и *условие инвариантности* (invariance condition).

Операции деструктора (Destructor operations)
Методы, разрушающие объекты и освобождающие занимаемую ими память.

Операции конструктора (Constructor operations)
Методы, которые создают и инициализируют состояние объекта.

Операции/методы доступа (Access operations/methods)
Операции/методы, осуществляющие доступ к состоянию объекта, но не изменяющие его.

Операция (Operation)
Процедура, которую объект умеет выполнять для предоставления доступа к своим данным. Операции реализуются как *методы* (method).

Операция-селектор (Selector operation)
Метод, который может осуществлять доступ к состоянию объекта, но не может изменять это состояние.

Ориентированный на классы/основанный на классах (Class-oriented/based)
Основанные на объектах системы, где каждый экземпляр принадлежит некоторому классу, но классы могут не иметь суперклассов.

Основанный на объектах (Object-based)
Системы считаются объектно-ориентированными, если объекты могут инкапсулировать данные и методы и обладают идентичностью.

Ограничитель (Hedge)
Термин из нечеткой логики, например `VERY` (очень) и `FIRLY` (довольно).

Открытый интерфейс (Public interface)
Часть объекта, методы или атрибуты, которые могут быть доступны через другие объекты. Открытая часть объекта образует его интерфейс.

Отложенное наследование (Deferred inheritance)
Использует классы с отложенной реализацией.

Отображение (Mapping)
Однозначная направленная ассоциация.

Отражение (Reflection)
Средство расширения модели на стадии выполнения, например, за счет использования *мощных типов* (power type).

Пакет (Package)
Конструкция языка Ada, объединяющая совокупность объявлений в одной программной единице, которая может быть использована для создания абстрактного типа данных.

Параметризованные классы/типы (Generic classes/types)
Классы/типы, для которых в качестве параметра задается тип переменной, например список.

Параметризованный пакет (Generic package)
Пакет в языке Ada, который содержит параметры настройки.

Пассивные объекты (Passive Objects)
См. *активные объекты* (active objects).

Перегрузка (Overloading)
Задание двух различных обязанностей двум различным сущностям, имеющим одно и то же имя.

Перегрузка операций (Operator overloading)
Специальный случай *полиморфизма* (polymorphism): присвоение нескольких значений одному и тому же символу операции. Термин "перегрузка" иногда применяется также для указания на использование одного и того же имени разными объектами.

Передача сообщений (Message passing)
Философия, которая заключается в том, что объекты могут взаимодействовать только посредством передачи сообщений, которые запрашивают выполнение некоторых операций.

Перекрытие (Overriding)
Возможность изменять в подклассе определение унаследованного метода или атрибута.

Переменная класса/атрибут класса (Class variable/class attribute)
Атрибут, который наследуется всеми подклассами и задает состояние набора экземпляров, а не поведение отдельных экземпляров. Значения переменных класса могут наследоваться и замещаться в подклассах.

Переменная/атрибут экземпляра (Instance variable/attribute)
Атрибут, содержащий данные экземпляра и описывающий его состояние. Подклассы могут наследовать только объявление переменной экземпляра, но не ее значение. См. также *атрибут* (attribute) и *переменная класса* (class variable).

Перманентность (Persistence)
Свойство объектов сохранять свою идентичность, состояние и описание в течение некоторого времени независимо от сеансов работы, во время которых эти объекты создаются или используются. Объекты хранятся готовыми к использованию на вспомогательном запоминающем устройстве.

Поведение (Behaviour)
Набор методов объекта, которые определяют его поведение.

Подкласс (Subclass)
Класс, который имеет связь АКО (это разновидность...) с более общим классом; как в предложении "СОБАКА — это вид МЛЕКОПИТАЮЩИХ".

Подсистема (Subsystem)
Общий термин, подобный терминам *уровень* (level), *предмет* (subject) или *предметная область* (subject area), который обозначает подраздел описания системы.

Подстановка каркаса (Framework substitution)
Ввод в каркас модели информации (имен типов) вместо символов шаблона.

Позднее связывание (Late binding)
То же, что и *динамическое связывание* (dynamic binding).

Полиморфизм (Polymorphism)
Дословно — множественность форм. Способность объекта или оператора ссылаться на экземпляры разных классов на стадии выполнения. Таким образом, полиморфные сообщения могут интерпретироваться по-разному, если они получены разными объектами или в разных контекстах.

Последовательное упорядочение расстояния (Inferential distance ordering)
Частичное упорядочение сети наследования в соответствии с числом суперклассов, через которые наследуется значение. Расстояние — это число шагов в цепочке наследования. Например, в правиле логического вывода "У Фидо есть шерсть, потому что Фидо — собака, а все собаки — млекопитающие (и млекопитающие имеют шерсть)" расстояние равно 2.

Построение обратной цепочки (Backward chaining)
Метод поиска (или наследования), который начинается с заключения и следует по цепочке правил в обратном направлении с целью обнаружения приемлемых причин или оснований для этого заключения.

Правило (Rule)
Правило или ограничение, которое определяет некоторое отношение между двумя или несколькими атрибутами и/или операциями в рамках объекта. Иногда правила задают стратегию разрешения конфликта, которая будет принята объектами, например, если возникает конфликт в результате множественного наследования.

Пред- и постусловия (Pre-conditions и post-conditions)
Логические утверждения, которые должны проверяться на истинность перед началом выполнения метода или после его завершения соответственно.

Предусловие (Guard)
Условие перехода на диаграмме переходов состояний.

Прецедент (Use case)
Связанная *последовательность* транзакций в диалоге с системой.

Примесь (Mixin)
Абстрактный класс, содержащий совокупность методов, которые могут наследоваться (смешиваться) любым классом. Обычно примеси связаны с системой CLOS объектно-ориентированного программирования на языке Common Lisp.

Программа (Programme)
Совокупность *проектов* (projects).

Продолжительные транзакции (Long transactions)
Транзакции, которые не подлежат аварийному завершению в случае взаимной блокировки. Для их реализации необходимо поддерживать контроль версий.

Проект (Project)
Совокупность действий, выполняемая в течение не более шести месяцев командой, состоящей из шести (или менее) человек, в результате чего получается система. Является частью *программы* (programme).

Прозрачность ссылок (Referential transparency)
Способ программирования, при котором каждое выражение или переменная имеет одно и то же значение в данной области видимости — все переменные являются локальными.

Производные зависимости/ассоциации (Derived dependencies/associations)
Зависимости или ассоциации между объектами, возникающие благодаря унаследованным характеристикам.

Производный класс (Derived class)
Уточнение или специализация существующего класса.

Протокол (Protocol)
Совокупность открытых операций объекта.

Прототип (Prototype)
Имитационная или экспериментальная версия компьютерной системы, применяемая с целью получения дополнительной информации о требованиях или проекте; либо экземпляр, действующий подобно классу в том смысле, что его свойства могут быть присвоены другим экземплярам; см. также *делегирование* (delegation).

Прямое построение цепочки (Forward chaining)
Метод поиска или логического вывода, который начинается с одного или нескольких фактов и, двигаясь вперед по цепочке правил, находит одно или несколько заключений.

Раннее связывание (Early binding)
То же, что *статическое связывание* (static binding), и противоположно по значению *позднему или динамическому связыванию* (late или dynamic binding).

Реализация (Implementation)
Закрытые или скрытые характеристики объекта.

Режим логического вывода (Inference regime)
Логика применения к процессу *набора правил* (ruleset), часто — исчисление предикатов первого порядка, включающее правила построения прямой и обратной цепочек.

Режимы (Modes)
Состояния системы (как правило, ее пользовательского интерфейса), которые препятствуют выполнению некоторых действий до тех пор, пока продолжается это состояние. Например, при открытом модальном диалоговом окне нельзя выполнять операции редактирования.

Самовызов (Self-reference)
То же, что и саморекурсия, — способность метода ссылаться на самого себя.

Саморекурсия (Self-recursion)
Способность отправлять сообщения самому себе.

Сборка мусора (Garbage collection)
Процесс освобождения памяти, которая была занята объектами, не подлежащими дальнейшему использованию или недоступными из текущего модуля.

Свойство (Feature)
Атрибут, операция, инвариант или набор правил, относящийся к типу.

Селектор (Selector)
Метод, который оценивает текущее состояние объекта.

Семантика данных (Data semantics)
Описание значения данных и их соотношений, особенно кратности отношений, модальности, наследования и композиции. Эти отношения включают ограничения целостности.

Сервер (Server)
Объект, на который воздействуют операции, но сам он не может влиять на другие объекты. Он может только отправлять сообщения другим объектам при получении от них запроса.

Сигнатура (Signature)
Интерфейс объекта минус его инварианты: его имя, атрибуты и методы.

Силлогизм (Syllogism)
Логический или некоторый другой процесс, в ходе которого связываются два несвязанных термина при помощи промежуточного термина. Отношение между частным, отдельным и универсальным (в любом порядке). Классический пример — это дедуктивный силлогизм: "Все люди смертны (универсальный). Гаусс — человек (частный). Поэтому Гаусс смертен (индивидуальный)".

Слабая типизация (Weak typing)
Противоположна по своему значению *сильной типизации* (strong typing). Ошибки, связанные с контролем типов, могут появляться на стадии выполнения.

Слот (Slot)
Это слово используется для обозначения атрибутов во фреймовых системах искусственного интеллекта. Заметим, что атрибуты фреймов могут содержать методы. Переменная экземпляра в CLOS (системе объектно-ориентированного программирования на языке Common Lisp).

Случайный объект (Accidental object)
Объект, определенный исключительно при помощи своих свойств. См. *существенный объект* (essential object) и *тавтологический объект* (tautological object).

Совместное поведение (Behaviour sharing)
Форма *полиморфизма* (polymorphism), при которой некоторые сущности имеют один и тот же интерфейс, что достигается посредством наследования или *перегрузки операций* (operator overloading).

Соглашение (Contract)
Набор сообщений, которые клиент может отправлять серверу, определяющий обязанности клиента. Иногда (в CRC-карточке класса) — связанное подмножество этого набора.

Сокрытие информации (Information hiding)
Принцип, гласящий, что состояние и реализация объекта или модуля должны быть закрыты и доступны только через их открытый интерфейс. Это понятие тесно связано с понятием *инкапсуляции* (encapsulation).

Сообщение (Message)
Запрос к объекту с целью выполнения одной из его операций.

Сотрудничество (Collaboration)
Отношение между клиентами и сервером, основанное на использовании или передаче сообщений.

Специализация (Specialization)
Это понятие противоположно по своему значению понятию *обобщения* (generalization).
Перемещение "вниз" по *структуре классификации* (classification structure).
Более ограниченный класс по сравнению с некоторыми другими классами, от которых он унаследован.

Специальный полиморфизм (Ad hoc polymorphism)
Присвоение одного и того же имени различным операциям, определенным для разных типов; при этом их семантика может быть различной.

Статическая типизация (Static typing)
Фиксация типов объектов на стадии компиляции.

Статическое связывание (Static binding)
Противоположно по значению динамическому связыванию. На стадии компиляции значения связываются с переменными или методы — с сообщениями. Называется также *ранним связыванием* (early binding).

Стереотип (Stereotype)
Метка, характеризующая роль, которую может играть класс или тип.

Строгая типизация (Strong typing)
Свойство языка программирования, гарантирующее, что все выражения будут совместимыми с точки зрения типов; обычно достигается посредством *статической типизации* (static typing).

Структура (Structure)
Совокупность зависимостей одного и того же типа на диаграмме UML, описывающей классы, типы или прецеденты; как правило, это зависимости одного из следующих типов: наследование, агрегация, ассоциация или использование.

Структура использования (Use structure)
Структура отношений между клиентами и серверами, связанными посредством передачи сообщений.

Структура классификации (Classification, AKO или IsA)
Структура в виде дерева или сети, основанная на семантических примитивах включения (AKindOf — разновидность) и принадлежности (IsA — это есть) и свидетельствующая о том, что при помощи наследования можно реализовать специализацию или обобщение. Объекты могут принимать участие не только в одной, но и в нескольких таких структурах, давая начало множественному наследованию. Строго говоря, классификация представляет собой отношение, существующее между экземплярами и их классом.

Структура композиции (Composition или A-Part-Of)
Структурированный в виде дерева набор связей, основанный на семантическом примитиве "целое-часть", который указывает на то, что определенные объекты могут быть собраны в рамках коллекции объектов. По своей природе объекты могут принимать участие не в одной, а в нескольких таких структурах. В UML композиция представляет собой ограниченную форму агрегации, где части создаются и разрушаются при помощи целого.

Суперкласс (Superclass)
Класс, который имеет один или несколько членов, которые сами по себе являются классами (более специальными). *Базовый класс* (base class).

Существенные объекты (Essential objects)
Объекты, известные не только благодаря значениям их атрибутов.

Сценарий (Scenario)
Конкретная реализация действия, задания, прецедента и др. Конкретное изложение некоторого бизнес-процесса.

Сценарий (Script)
Стереотипное, общее и существенное описание действия (задания, прецедента, вида деятельности и т.д.).

Тавтология (Tautology)
Утверждение, которое является истинным по определению, например "красные розы являются красными". Тавтологические объекты не имеют никаких других характеристик, кроме тех, которые участвуют в описании, например RedThings.

Тип (Type)
Набор данных с определенными над ним операциями. См. *абстрактный тип данных* (abstract data type).

Тип данных (Data type)
Тип (type) без операций.

Тип/класс ассоциации (Association type/class)
Задание типа/класса ассоциации с целью описания ее деталей.

Типы состояний (Stative types)
Типы, которые представляют состояния объекта или объектов.

Триггер (Trigger)
Демон (demon). Правило, которое запускается в ответ на изменение состояния.

Триггер базы данных (Database trigger)
См. *триггер* (trigger).

Универсальность (Genericity)
Возможность параметризации классов или типов; специальный случай *полиморфизма* (polymorphism). *Ограниченная универсальность* (constrained genericity) требует параметров для специальных операций, в противном случае универсальность считается *неограниченной* (unconstrained).

Универсальный язык моделирования UML
Универсальный язык моделирования, предназначенный для объектно-ориентированного анализа и проектирования; он принят в качестве стандарта рабочей группой по развитию стандартов объектного программирования OMG и широко рассматривается в рамках данной книги.

Условие инвариантности (Invariance condition)
Условие, которое остается истинным на протяжении всего времени выполнения метода.

Утверждения (Assertions)
Утверждения вида "A — это B" в языках, основанных на правилах. Называются также "фактами"; либо в формальных языках — общий термин для определения *предусловий* (pre-condition), *постусловий* (post-condition) или *условий инвариантности* (invariance-condition).

Уточняющие методы (Elaborational methods)
Действие этих методов заключается в том, что по мере разработки общая объектная модель дополняется деталями. См. также *методы преобразования* (translational method).

Фацет (Facet)
Характеристика свойства объекта.

Феноменология (Phenomenology)
Школа мышления, существующая в рамках современной философии, которая провозглашает активную роль предмета, характеризуемую следующим девизом: нет объекта без предмета. Этот термин используется для строгой ссылки на идеалистические позиции Брентано (Brentano) и Хассерла (Husserl) и их последователей экзистенциалистов — Хейдеггера (Heidegger) и Сартра (Sartre) или, в более широком смысле, для включения Гегеля (Hegel) и некоторых современных материалистов.

Фрагмент памяти (Chunk)
Участок активной памяти, используемый как единица информации.

Фрейм (Frame)
Структура данных, используемая в области искусственного интеллекта подобно объекту. Фреймы имеют *слоты* (slots), которые могут содержать данные и процедуры.

Функциональная семантика (Functional semantics)
Описание значения процедурных или управленческих аспектов объекта.
В частности, описание правил для триггеров, способов разрешения конфликтов, обработки исключительных ситуаций, режимов управления и общих бизнес-правил.

Функция принадлежности (Membership function)
Характеристическая функция множества (или нечеткого множества), которая принимает значения в интервале от 0 до 1. Значение 0 говорит о том, что элемент не принадлежит множеству, а 1 свидетельствует о полной принадлежности. В нечетких множествах, например множество высоких людей, элемент может частично принадлежать множеству. Например, человеку ростом 5 футов 11 дюймов во множестве высоких людей может быть присвоено значение 0,6. С точки зрения любого практического применения функция принадлежности соответствует нечеткому множеству.

Функция-член класса (Member function)
Функция, объявленная внутри класса и имеющая доступ к его внутреннему состоянию.

Целостность ссылок (Referential integrity)
Свойство реляционных баз данных, которое гласит следующее: если отношение R содержит внешний ключ, значение которого берется из значений первичного ключа отношения S, тогда каждое вхождение внешнего ключа в отношении R должно либо совпадать со значением первичного ключа в отношении R, либо равняться нулю. R и S могут представлять одно и то же отношение. В более общем смысле это свойство пары слабых полуобратных ассоциаций между объектами, которое означает следующее: если вы прослеживаете обе ассоциации, то пересечение отображений начальных значений будет содержать эти величины.

Частичное наследование (Partial inheritance)
См. *нечеткое наследование* (fuzzy inheritance).

Шаблон (Pattern)
Стандартное именованное решение повторяющейся проблемы. Имя шаблона позволяет специалистам без труда обсуждать его и применять при проектировании.

Шаблон (Template)
См. *параметризованный тип* (generic type) и *шаблон модели* (model template).

Шаблон модели (Model template)
См. *каркас модели* (model framework).

Эволюция схемы (Schema evolution)
Изменение структуры базы данных.

Экземпляр (Instance)
Конкретный объект или отдельный экземпляр класса.

Эмпиризм (Empiricism)
Философский принцип, согласно которому первичным является сознание, а не объективная реальность. Эмпиризм и некоторые происходящие от него философские течения, например учения Канта, логического позитивизма и прагматизма, являлись господствующей философией в западной науке, начиная со времен идей Бэкона.

Эпистемология (Epistemology)
Наука или теория о знаниях.

Эффект (Effect)
Постусловие, которое объединяется вместе со всеми другими постусловиями операции некоторого типа.

Язык агентного взаимодействия (Agent communication language — ACL)
Язык, используемый разнородными мобильными и интеллектуальными агентами для обмена данными, планами и т.д.

Библиография

1. Abadi M. and Cardelli L. *A Theory of Objects*. Berlin: Springer-Verlag, 1996.
2. Abbott R.J. *Program Design by Informal English Descriptions*. Communications of the ACM 26(11), 1983, p. 882–894.
3. Abrial J.R. *Data Semantics*. In Klimbie and Koffeman (Eds.) Data Base Management. Amsterdam: North-Holland, 1974.
4. Ackroyd M. and Daum D. *Graphical notation for object-oriented design and programming*. Journal Of Object-Oriented Programming 3(5), 1991, p. 18–28.
5. Adams G. and Corriveau J-P. *Capturing Object Interactions*. In Magnusson, Meyer, Nerson and Perrot, 1994.
6. Agha G. *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge MA: MIT Press, 1986.
7. Agha G. and Hewitt C. *Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming*. In Shriver B. and Wegner P. (Eds.), 1987.
8. Agha G., Wegner P. and Yonezawa A. (Eds.) *Research Directions in Concurrent Object-Oriented Programming*. MIT Press, 1993.
9. Agha S. *Hypertext Systems*. M.Sc. Thesis, City University, London, 1989.
10. Agrawal R. and Gehani N. *ODE: the language and the data model*. Proc. ACM SIGMOD Conf. on the Management of Data. Portland, Oregon, 1989.
11. Ahad R. and Dedo D. *OpenODB from Hewlett-Packard: a commercial object-oriented database management system*. Journal of Object-Oriented Programming. 4(9), 1992, p. 31–35.
12. Ahmed S. et al. *Object-oriented database management systems for engineering: a comparison*. Journal Of Object-Oriented Programming 5(3), 1992, p. 27–44.
13. Akscyn R.M. and McCracken D.L. *The ZOG Approach to Database Management*. In Proceedings of the Trends and Applications Conference: Making Database Work, Gaithersburg, Maryland, 1984.
14. Akscyn R.M., McCracken D.L. and Yoder E.A. *KMS: A Distributed Hypermedia System for Managing Knowledge in Organisations*. Communications of the ACM, 31(7), 1988, p. 820–835.
15. Alabiso B. *Transformation of data flow analysis models to object-oriented design*. In Meyrowitz, 1988.
16. Albano A. et al. *The Type System of Galileo*. In Atkinson M.P., Buneman O.P. and Morrison R. (Eds.) Data Types and Persistence. Berlin: Springer-Verlag, 1988, p. 101–119.
17. Albrecht A.J. *Measuring application developement productivity*. Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium, October 1979, p. 34–43.

18. Albrecht A.J. and Gaffney J.E. *Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation.* IEEE Transactions on Software Engineering, 9(6), 1983, p. 639–647.
19. Aleksander I. and Morton H. *An Introduction to Neural Computing.* Chapman & Hall, 1990.
20. Alencar A.J. and Goguen J.A. *OOZE: An object-oriented Z environment.* In America P. (Ed.) Proc. of ECOOP'91. Lecture Notes in Computer Science 512, Berlin: Springer, 1991.
21. Alexander C. *Notes on the Synthesis of Form.* Harvard: University Press, 1964.
22. Alexander C. *The Timeless Way of Building.* Oxford: University Press, 1979.
23. Alexander C., Ishikawa S. and Silverstein M. *A Pattern Language.* Oxford: University Press, 1977.
24. Alexander C. *A Foreshadowing of 21st Century Art.* New York: Oxford University Press, 1996.
25. Alexander C. *The Origins of Pattern Theory: The Future of the Theory and the Generation of a Living World.* IEEE Software September/October, 1999, p. 71–82.
26. Alpert S., Brown K. and Woolf B.R. *The Design Patterns Smalltalk Companion.* Reading MA: Addison-Wesley, 1998.
27. Allen P. and Frost S. *Component-Based development for Enterprise Systems: Applying the SELECT Perspective.* Cambridge: University Press/SIGS, 1998.
28. Altham J.E.J. *The Logic of Plurality.* Methuen, 1971.
29. Ambler S. *Process Patterns.* Cambridge, England: The University Press, 1998.
30. Ambler S. *More Process Patterns.* Cambridge, England: The University Press, 1999.
31. Andersen B. *Ellie: a general, fine-grained, first-class, object based language.* Journal Of Object-Oriented Programming 5(2), 1992, p. 35–41.
32. Anderson J.A., McDonald J., Holland L. and Scranage E. *Automated Object-Oriented Requirements Analysis and Design.* Proceedings of the 6th Washington ADA Symposium, 1989, p. 265–272.
33. Anderson J.R. *Language, Memory and Thought.* Laurence Erlbaum, 1976.
34. Andleigh P.K. and Gretzinger M.R. *Distributed Object-Oriented Data-Systems Design.* Englewood Cliffs NJ: Prentice Hall, 1992.
35. Andrews T. and Harris C. *Combining Language and Database Advances in an Object-Oriented Development.* In Zdonik and Maier, 1990.
36. Aoyama M. *Concurrent Development Process Model.* IEEE Software, July 1993, p. 46–55.
37. Apple. *The MacApp Interim Manual.* Cupertino CA: Apple Computer Inc., 1988.
38. Apple Computer Inc. *Apple Human Interface Guidelines: The Apple Desktop Interface.* Cupertino, CA: Addison-Wesley, 1987.
39. Arranga E.C. and Coyle F.P. *Object-Oriented COBOL.* New York: SIGS Books, 1996.
40. Arthur L.J. *Measuring Programmer Productivity and Software Quality.* New York: Wiley, 1985.

41. Ashby W.R. *An Introduction to Cybernetics*. London: Chapman & Hall, 1956.
42. Ashby W.R. *Design for a Brain. 2nd Edition*. London: Chapman & Hall, 1960.
43. Atkinson M.P. and Buneman O.P. *Types and Persistence in Database Programming Languages*. ACM Computing Surveys 19(2), 1988, p. 105–190.
44. Atkinson M.P., Bancilhon F., DeWitt D., Dittrich K., Maier D. and Zdonik S. The Object-Oriented Database System Manifesto. *Deductive and Object-Oriented Databases*. Amsterdam: Elsevier 1990; also in *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 4–6, 1989, p. 40–57.
45. Attwood T. *At last! A distributed database for Windows 3.0*. Object Magazine 1(1), 1991, p. 36–57.
46. Austin J.L. *How to Do Things with Words*. Cambridge MA: Harvard University Press, 1962.
47. Bachman C. *The role concept in data models*. In Proceedings of the 3rd International Conference on Very Large Databases, IEEE, New York, 1977, p. 464–476.
48. Baecker R.M. and Buxton W.A.S. (Eds.) *Readings in Human Computer Interaction: A multi-disciplinary approach*. San Mateo CA: Morgan Kaufmann, 1987.
49. Bailin S.C. *An object-oriented requirements specification method*. Comm. ACM 32(5), 1989, p. 608–623.
50. Baldwin J.F. and Martin T.P. *Fuzzy classes in object-oriented logic programming*. Proc. 5th IEEE International Conf. on Fuzzy Systems. New Orleans LA, 1996, p. 1358–1365.
51. Bancilhon F., Delobel C. and Kanillakis P. (Eds.) *Building an Object-Oriented Database System: The Story of O_2*. Morgan-Kaufmann, 1991.
52. Banker R.D., Kauffman R.J. and Zweig D. *Repository Evaluation of Software Reuse*. IEEE Trans. on Software Engineering 19(4), 1993.
53. Bapat S. *Object-Oriented Networks: Models for architecture, operations and management*. Englewood Cliffs NJ: Prentice Hall, 1994.
54. Barfield L. *The User Interface: Concepts & Design*. Wokingham: Addison-Wesley, 1993.
55. Barker R. *CASE*METHOD: Entity Relationship Modelling*. Wokingham: Addison-Wesley, 1990.
56. Barr A. and Feigenbaum E. *The Handbook of Artificial Intelligence* (3 vols.). Pitman, 1981.
57. Basden A. *Towards a Methodology for Building Expert Systems I*. Codex 2(1) 15–19, Uxbridge: Creative Logic Ltd, 1990.
58. Basden A. *Towards a Methodology for Building Expert Systems II*. Codex 2(2) 19–23, Uxbridge: Creative Logic Ltd, 1990.
59. Bass L., Clements P. and Kazman R. *Software Architecture in Practice*. Reading MA: Addison-Wesley, 1998.
60. Bassett P.G. *Framing Software Reuse*. Upper Saddle River NJ: Prentice Hall, 1997.
61. Beck K. *A Short Introduction to Pattern Languages*. Smalltalk Report February, 1993.
62. Beck K. *Smalltalk Best Practice Patterns*. Upper Saddle River NJ: Prentice Hall, 1997.
63. Beck K. *Extreme Programming Explained: embrace change*. Reading MA: Addison-Wesley, 2000.

64. Beck K. and Cunningham W. *A Laboratory for Teaching Object-Oriented Thinking.* In Meyrowitz, 1989.

65. Beck K. and Fowler M. *Planning Extreme Programming.* Reading MA: Addison-Wesley, 2000.

66. Beech D. *Groundwork for an Object-Oriented Database Model.* In Shriver and Wegner, 1987.

67. Beech D. *Relational versus Object DBMS, lecture notes, Object World.* San Francisco, July 1992.

68. Belcher K. *Object-orientation: The COBOL approach.* Object Magazine 1(1), 1991, p. 74–83.

69. Bell D.A., Shao J. and Hull M.E.C. *Integrated Deductive Database System Implementation: A Systematic Study.* Computer Journal 33(1), 1990, p. 40–48.

70. Bellman R. and Zadeh L.A. *Decision Making in a Fuzzy Environment.* Management Science 17(4), 1970, p. 141–164.

71. Berard E.V. *Essays on Object-Oriented Software Engineering — Volume 1.* Englewood Cliffs NJ: Prentice Hall, 1993.

72. Berliner H. *The B* Tree Search Algorithm: A Best-first Proof Procedure.* In Webber B.L. and Nilsson N.J. (Eds.) Readings in Artificial Intelligence. Tioga Publishing Company, 1981.

73. Beyer H. and Holtzblatt K. *Contextual Design: Defining Customer-Centred Systems.* New York: Morgan Kaufmann, 1997.

74. Bezivin J. and Meyer B. *TOOLS4: Proceedings of the fourth international conference on the Technology of Object-Oriented Languages and Systems.* Englewood Cliffs NJ: Prentice Hall, 1991.

75. Bezivin J., Hullot J-M., Cointe P. and Leiberman H. (Eds.) *ECOOP'87 European Conference on object-oriented programming.* Lecture Notes in Computer Science 276, Berlin: Springer Verlag, 1987.

76. Biggerstaff T. and Richter C. *Re-usability Framework, Assessment and Directions. Tutorial: Software Reuse — Emerging Technology.* IEEE Computer Society, EH0278-2, 1989, p. 3–11.

77. Bigus J.P. and Bigus J. *Constructing Intelligent Agents with JAVA.* New York: Wiley, 1998.

78. Birrel N.D. and Ould M.A. *A Practical Handbook for Software Development.* Cambridge: University Press, 1985.

79. Birtwistle G.M. *Discrete Event Modelling On Simula.* London: MacMillan, 1979.

80. Blaauw G.A. *Hardware Requirements for the Fourth Generation.* In Greunberger F. (Ed.) Fourth Generation Computers. Englewood Cliffs NJ: Prentice Hall, 1970.

81. Black A., Hutchinson N., Jul E. and Levy H. *Object Structure in the Emerald System.* In Meyrowitz, 1986, p. 78–87.

82. Blaha M. and Premerlani W. *Object-Oriented Modelling and Design for Database Applications.* Upper Saddle River NJ: Prentice Hall, 1998.

83. Blair G., Gallagher J., Hutchison D. and Shepherd D. *Object-Oriented Languages, Systems and Applications*. London: Pitman, 1991.

84. Blum A. *Neural Networks in C++: An object-oriented framework for building connectionist systems*. New York: Wiley, 1992.

85. Blum B.I. *Beyond Programming: To a New Era of Design*. New York: Oxford University Press, 1996.

86. Blum B.I. *Software Engineering: A Holistic Approach*. New York: Oxford University Press, 1998.

87. Boar B. *Application Prototyping: A Requirements Definition Strategy for the 80s*. New York: Wiley, 1984.

88. Bobrow D. G. and Stefik M. *The LOOPS Manual*. Xerox Corporation, 1983.

89. Bobrow D. G. and Winograd T. *An Overview of KRL, A Knowledge Representation Language*. Cognitive Science 1(1), 1977, p. 3–46. Also in Brachman and Levesque, 1985.

90. Bobrow D., Kahn K., Kiczales G., Masiuter L., Stefik M. and Zdybel F. *CommonLOOPS: Merging Lisp and object-oriented programming*. In Meyrowitz, 1986, p. 17–29.

91. Bodkin T. and Graham I.M. *Case Studies of Expert Systems Development using Microcomputer Software Packages*. Expert Systems 6(1), 1989, p. 12–16.

92. Boehm B.W. *Software Engineering Economics*. Englewood Cliffs NJ: Prentice Hall, 1981.

93. Booch G. *Object-Oriented Design*. Ada Letters 1(3), 1982, p. 64–76.

94. Booch G. *Object-Oriented Development*. IEEE Trans. on Software Eng., vol SE-12(2), 1986, p. 211–221.

95. Booch G. *Software Engineering with ADA (2nd Edition)*. Benjamin Cummings, 1987.

96. Booch G. *Software Components with ADA*. Benjamin Cummings, 1987.

97. Booch G. *On the Concepts of Object-Oriented Design*. In Ng P.A. and Yeh R.T. (Eds.) Modern Software Engineering: Foundations and Current Perspectives. Van Nostrand, New York, 1990, p. 165–204.

98. Booch G. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

99. Booch G. *Object Oriented Design with Applications, 2nd Edition*. CA: Benjamin/Cummings, 1994.

100. Booch G. *Software Architecture and the UML*, keynote presentation to UML World, 1999.

101. Booch G., Rumbaugh J. and Jacobson I. *The Unified Modelling Language User Guide*. Reading MA: Addison-Wesley, 1999.

102. Boose J.H. *Expertise Transfer for Expert Systems Design*. Amsterdam: Elsevier, 1986.

103. Boose J.H. and Bradshaw J.M. *Expertise Transfer and Complex Problems: Using AQUINAS as Knowledge Acquisition Workbench for Knowledge Based Systems*. In Boose J.H. and Gaines B.R. (Eds.) Knowledge Acquisition Tools for Expert Systems. Academic Press, 1988.

104. Boose J.H. and Gaines B.R. (Eds.) *Knowledge Acquisition for Knowledge Based Systems*. Academic Press, 1988.

105. Borenstein N.S. *Multimedia Applications Development with the Andrew Toolkit*. Englewood Cliffs NJ: Prentice Hall, 1990.

106. Borenstein N.S. *Programming as if People Mattered: Friendly Programs, Software Engineering, and Other Noble Delusions*. Princeton NJ: Princeton University Press, 1991.

107. Borning A. and Ingalls D. *A type declaration and inference system for Smalltalk*. In Proc. of 9th Annual ACM Symposium on Principles of Programming Languages. Albuquerque, NMex, January, New York: ACM, 1982, p. 133–141.

108. Bosch J. *Design and Use of Software Architectures*. Harlow, England: Addison-Wesley, 2000.

109. Brachman R.J. *What IS-A is and isn't: an analysis of taxonomic links in semantic networks*. IEEE Computer 16(10), 1983, p. 30–36.

110. Brachman R.J. and Levesque H.J. *Readings in Knowledge Representation*. Morgan Kaufmann, 1985.

111. Bradley N. *The XSL Companion*. Harlow, England: Addison-Wesley, 2000.

112. Braithwaite R.B. *Scientific Explanation*. Cambridge: University Press, 1953.

113. Braune R. and Foshay W.R. *Towards a practical model of cognitive information processing, task analysis and schema acquisition for complex problem solving situations*. Instructional Science 12, 1983, p. 121–145.

114. Bretl R., Maier D., Otis A., Penney J. *et al.* The *GemStone Data Management System*. In Kim W. and Lochovsky F.H. (Eds.), 1989.

115. Brice A. *Using models in analysis*. Computing 27th May 1993, p. 41.

116. Brodie M.L. and Mylopoulos J. (Eds.) *On Knowledge Base Management Systems*. Berlin: Springer-Verlag, 1986.

117. Brodie M.L., Mylopoulos J. and Schmidt J.W. (Eds.) *On Conceptual Modelling*. Berlin: Springer-Verlag, 1984.

118. Brooks F. *The Mythical Man Month*. Reading MA: Addison-Wesley, 1975.

119. Brooks F. *No Silver Bullet: Essence and Accidents of Software Engineering*. In H.-J. Kluger (Ed.) Information Processing '86. Amsterdam: Elsevier, 1986.

120. Brown A.W. *Object-Oriented Databases and their Applications to Software Engineering*. London: McGraw Hill, 1991.

121. Brown R.G. *Adding business rules to data models*. Data Base Newsletter 20(6), 1992.

122. Brown W. Object-Oriented Testing, Proceedings of Object Technology 93, BCS OOPS Group annual conference, 1993.

123. Brown W., Malveau R., McCormick H. and Mowbray T. *AntiPatterns: Refactoring Software Architectures and Project in Crisis*. New York: Wiley, 1998.

124. Browne D. *STUDIO: STructured User interface Design for Interaction Optimisation*. London: Prentice Hall, 1994.

125. Bruce T.A. *Simplicity and complexity in the Zachman framework*. Database Newsletter 20(3), 1992, p. 3–11.

126. Brynjolfsson E. *The productivity paradox in information technology*. Communications of the ACM 36(12), 1993, p. 67–77.

127. Buchanan B. and Shortliffe E. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Reading MA: Addison-Wesley, 1984.

128. Budd T. *Understanding Object-Oriented Programming with JAVA*. Reading MA: Addison-Wesley, 1998.

129. Buhr R.J.A. *System Design with Ada*. Englewood Cliffs NJ: Prentice Hall, 1984.

130. Buhr R.J.A. and Casselman R.S. *Use Case Maps for Object-Oriented Systems*. Englewood Cliffs NJ: Prentice Hall, 1996.

131. Bulman D. *Refining candidate objects*. Computer Language, January 1991, p. 30–39.

132. Burstall R.M., McQueen D.B. and Sannella D.T. *HOPE: An experimental applicative language*. Internal Report CSR-62-80, University of Edinburgh, 1980.

133. Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M. *Pattern-oriented Software Architecture: A System of Patterns*. Chichester, England: Wiley, 1996.

134. Bush V. *As We May Think*. Atlantic Monthly 176(1), 1945, p. 101–108.

135. Butler A. and Chamberlain G. *The ARIES Club: Experience Of Expert Systems In Insurance and Investment*. In Moralee D.S. (Ed.) Research And Development In Expert Systems IV. Cambridge University Press, 1988.

136. Cain B. and Coplien J.O. *A rôle-based empirical process modeling environment*. Proc. 2nd Int. Conf. on Software Process, Berlin, 1993.

137. Cameron J. *Ingredients for a New Object-Oriented Method: Development process, shared object modelling and GUI design will integrate OT and MIS*. Object Magazine 2(4), 1992, p. 64–67.

138. Cant S.N., Jeffrey D.R. and Henderson-Sellers B. *A conceptual model of cognitive complexity of elements of the programming process*. Inf. Software Technology (to appear), 1992.

139. Card D.N. and Glass R.L. *Measuring Software Design Quality*. Englewood Cliffs NJ: Prentice Hall, 1990.

140. Card S.K., Moran T.P. and Newell A. *The Psychology of Human Computer Interaction*. Hillsdale NJ: Lawrence Erlbaum, 1983.

141. Cardelli L. and Wegner P. *On Understanding Types, Data Abstraction and Polymorphism*. ACM Computing Surveys 17(4), 1985, p. 471–522.

142. Cardenas A. and McLeod D. (Eds.) *Research Foundations on Object-Oriented Databases*. Englewood Cliffs NJ: Prentice Hall, 1990.

143. Carey J., Carlson B. and Graser T. *San Francisco Design Patterns*. Reading MA: Addison-Wesley, 2000.

144. Carmichael A. *Approaches to Object-Oriented Analysis and Design*. London: Gower Press, 1994.

145. Carrington D., Duke D., Duke R., King P., Rose G. and Smith G. *Object-Z: An object-oriented extension to Z*. In Formal Description Techniques II. Amsterdam: North-Holland, 1990.

146. Carroll J.M. *Scenario-Based Design*. Chichester, England: Wiley, 1995.

147. Cato J. *User Centred Interface Design*. Harlow, England: Addison-Wesley, 2000.

148. Cattell R.G.G. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Reading MA: Addison-Wesley, 1991.

149. Cattell R.G.G. and Skeen J. Engineering Database Benchmark. Technical Report. Sun Microsystems, Mountain View CA, 1990.

150. Cattell R. (Ed.) *The Object Database Standard: ODMG-93*. San Mateo, CA: Morgan Kaufmann Publishers, 1994.

151. Champeaux D. de. *Object-Oriented Development Process and Metrics*. Upper Saddle River NJ: Prentice Hall, 1997.

152. Champeaux D. de, Lea D. and Faure P. *Object-Oriented System Development*. Reading MA: Addison-Wesley, 1993.

153. Chan A., Dayal U. Fox S. and Ries D. *Supporting a Semantic Data Model in a Distributed Database System*. In Proceedings of the 9th International Conference on Very Large Databases. Very Large Database Endowment, Saratoga, CA, 1983, p. 354–363.

154. Charniak E. and McDermott D. *Introduction to Artificial Intelligence*. Reading MA: Addison-Wesley, 1985.

155. Chaudhri A.B. and Osmon P. *A Comparative Evaluation of the Major Commercial Object and Object-Relational DBMSs*. Technical Report, London, England: City University, 1996.

156. Checkland P. *Systems Thinking, Systems Practice*. Chichester: Wiley, 1981.

157. Checkland P. and Scholes J. *Soft Systems Methodology in Action*. Chichester, England: Wiley, 1991.

158. Cheesman J. and Daniels J. *Component Modeling with UML*. Harlow, England: Addison-Wesley, 2000.

159. Chen P. *The Entity-Relationship Model: Toward a Unified View of Data*. ACM Transactions on Database Systems 1(1), 1976, p. 9–36.

160. Cheng J. and Jones C. *On the usability of logics which handle partial functions*. In Morgan C and Woodcock J. (Eds.) Proc. 3rd Refinement Workshop. Berlin: Springer, 1990.

161. Chidamber S.R. and Kemerer C.F. *Towards a metrics suite for object-oriented design*. In Paepcke A. (Ed.) OOPSLA'91 ACM Conference on Object-Oriented Programming Systems, Languages and Applications. Reading MA: Addison-Wesley, 1991.

162. Chidamber S.R. and Kemerer C.F. *A metrics suite for object-oriented design*. CISR Working Paper 249, MIT Sloan School of Management, Cambridge MA, 1993.

163. Chidamber S.R. and Kemerer C.F. *A metrics suite for object-oriented design*. IEEE Trans. Software Engineering 20, 1994, p. 476–493.

164. Chomsky N. *Rules and Representations*. Oxford: Basil Blackwel! , 1980.

165. Chorafas D.N. and Steinmann H. *Object-Oriented Databases: An Introduction*. Englewood Cliffs NJ: Prentice Hall, 1993.

166. Clausewitz C. von. *On War*. Rapoport translation of 1908, Harmondsworth: Penguin Books, 1968.

167. Coad P. *Object-Oriented Patterns*. Comms. ACM 35(9), 1992, p. 152–158.

168. Coad P., LeFebvre E. and DeLuca J. *Java Modeling in Color with UML*. Upper Saddle River NJ: Prentice Hall, 1999.

169. Coad P. and Nicola J. *Object-Oriented Programming*. Englewood Cliffs NJ: Yourdon Press/Prentice Hall, 1993.

170. Coad P., North D. and Mayfield M. *Object Models: Strategies, Patterns and Applications*. Upper Saddle River NJ: Prentice Hall, 1997.

171. Coad P. and Yourdon E. *Object-Oriented Analysis. 1st Edition*. Englewood Cliffs NJ: Yourdon Press/Prentice Hall, 1990.

172. Coad P. and Yourdon E. *Object-Oriented Analysis. 2nd Edition*. Englewood Cliffs NJ: Yourdon Press/Prentice Hall, 1991.

173. Coad P. and Yourdon E. *Object-Oriented Design*. Englewood Cliffs NJ: Yourdon Press/Prentice Hall, 1991.

174. Cockburn A. *Using goal-based use cases*. J. Object-Oriented Programming 10(5), 1997.

175. Codd E.F. *A Relational Model of Data for Large Shared Data Banks*. Comm. of the ACM 13(6), 1976, p. 377–387.

176. Codd E.F. *Is Your Relational Database Management System Really Relational?*. Oracle Users' Conference, San Diego, California, 1985.

177. Colbert E. *The OOSD Method: Requirements analysis with object-oriented software development*. In Carmichael, 1994.

178. Coleman D. and Hayes F. *Lessons from Hewlett-Packard's experience of using object-oriented technology*. In Bezivin and Meyer, 1991.

179. Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F. and Jeremaes P. *Object-Oriented Development: The Fusion Method*. Hemel Hempstead, England: Prentice Hall, 1994.

180. Collins H.M. *Artificial Experts: Social knowledge and intelligent machines*. Cambridge MA, MIT Press, 1990.

181. Comer D.E. *Computer Networks and Internets. 2nd Edition*. Upper Saddle River NJ: Prentice Hall, 1999.

182. Connell J.L and Shafer L.B. *Structured Rapid Prototyping: An Evolutionary Approach*. Yourdon Press, 1989.

183. Constantine L.L. *Essential modeling: use cases for user interfaces*. Interactions (ACM), 2(2), 1995.

184. Constantine L.L. and Lockwood L. *Software for Use: Models and Methods of Usage-Centred Design*. Reading MA: Addison-Wesley, 1999.

185. Cook S. (Ed.) *ECOOP'89 European Conference on Object-Oriented Programming*. Cambridge University Press, 1989.

186. Cook S. *Analysis, Design, Programming: What's the Difference?* In O'Callaghan A.J. and Leigh M. (Eds.) Object Technology Transfer. Henley-on-Thames: Alfred Waller, 1994.

187. Cook S. and Daniels J. *Essential techniques for object-oriented design*. Proceedings of OOPS-59. London: BCS OOPS Group, 1992.

188. Cook S. and Daniels J. *Designing Object Systems*. Hemel Hempstead, England: Prentice Hall, 1994.

189. Cook W. *A Denotational Semantics of Inheritance and its Correctness.* In Meyrowitz (1989), 1989.

190. Cooper J.W. *Java Design Patterns.* Reading MA: Addison-Wesley, 2000.

191. Coplien J.O. *Advanced C++: Programming Styles and Idioms.* Reading MA: Addison-Wesley, 1992.

192. Coplien J.O. *A Generative Development-Process Pattern Language.* In Coplien and Schmidt, 1995.

193. Coplien J.O. *Reevaluating the Architectural Metaphor: Toward Piecemeal Growth.* IEEE Software September/October, 1999, p. 40–44.

194. Coplien J.O. and Schmidt D. (Eds.) *Pattern Languages of Program Design.* Reading NJ: Addison-Wesley, 1995.

195. Costello J. The shape of screens to come. *Computer Weekly*, 27 August, 1992.

196. Coutaz J. *PAC, an Object-Oriented Model for Implementing User Interfaces.* Laboratoire de Genie Informatique, University de Grenoble, BP 68, 1987.

197. Cox B.J. *Object-Oriented Programming — An Evolutionary Approach.* Reading MA: Addison-Wesley, 1986.

198. Cox B.J. and Novobilski A. *Object-Oriented Programming – An Evolutionary Approach. 2nd Edition.* Reading MA: Addison-Wesley, 1991.

199. Cox E. *The Fuzzy Systems Handbook: A practitioner's guide to building, using and maintaining fuzzy systems.* Boston: Academic Press, 1994.

200. Cross V. *A unifying framework for the fuzzy object model.* Proc. 5th IEEE International Conf. on Fuzzy Systems, New Orleans LA, 1996, p. 1358–1365.

201. Cross V. and Firat A. *Fuzzy objects for geographic information system.* Fuzzy Sets and Systems 113(1), 2000, p. 19–36.

202. D'Souza D.F. *Dynamic Modelling of Object-Oriented Systems.* Proceedings of OOP'93/C++ World, Munchen 1993, New York: SIGS Publications, 1992.

203. D'Souza D.F. *Framework and Component Based Design.* Tutorial Notes, Object Expo Europe '97, 1997.

204. D'Souza D.F. and Wills A.C. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Reading MA: Addison-Wesley, 1999.

205. Dadam P. *Research and Development Trends in Relational Databases (NF2 Relations).* IBM Scientic Centre, Heidelberg, 1988.

206. Dahl O.J. *Object-Oriented Specification.* In Shriver and Wegner 1987, p. 561–576.

207. Dahl O.J. and Nygaard K. *SIMULA — An Algol-based Simulation Language.* Comm. of the ACM 9, 1966, p. 671–678.

208. Dahl O.J., Myrhaug B. and Nygaard K. *SIMULA 67 Common Base Language.* Norwegian Computing Centre, Oslo, 1968.

209. Dalton R. *Predicting the Cost of a Sale.* Software Management, March 1992.

210. Danforth S. and Tomlinson C. *Type Theories and Object-Oriented Programming.* ACM Computing Surveys 20(1), 1988, p. 29–72.

211. Daniels J. and Cook S. *Strategies for sharing objects in distributed systems*. JOOP, 5(8), 1993, p. 27–36.

212. Daniels J. Component contracts, keynote lecture at TOOLS Europe, 2000.

213. Daniels P.J. *The user modelling function of an intelligent interface for document retrieval systems*. In Brookes B.C. (Ed.) IRFIS 6: Intelligent Information Systems for the Information Society. Amsterdam: North Holland, 1986, p. 162–176.

214. Date C.J. *An Introduction to Database Systems. 3rd Edition*. Reading MA: Addison-Wesley, 1981.

215. Date C.J. *An Introduction to Database Systems. Volume II*. Reading MA: Addison-Wesley, 1983.

216. Davenport T.H. *Process Innovation: Reengineering work through information technology*. Harvard: Business School Press, 1993.

217. Davenport T.H. and Short J.E. *The New Industrial Engineering: Information Technology and Business Process Redesign*. Sloan Management Review, Summer 1990, p. 11–27.

218. Davis A.M. *Software Requirements: Objects, Functions and States, Revision*. Englewood Cliffs NJ: Prentice Hall, 1993.

219. Davis R. and Lenat D.B. *Knowledge Based Systems in Artificial Intelligence*. NY: McGraw Hill, 1982.

220. Dedene G. *M.E.R.O.D.E. and the practical realization of object-oriented business models*. In Carmichael, 1994.

221. Deitel H.M. and Deitel P.J. *C++ How to Program*. Englewood Cliffs NJ: Prentice Hall, 1994.

222. Delobel C., Lécluse C. and Richard P. *Databases: From Relational to Object-Oriented Systems*. International Thompson Publishing, 1995.

223. DeMarco T. *Controlling Software Projects*. Yourdon Press, 1982.

224. Demers A.J. and Donahue J.E. *Revised report on Russell*. TR79-389, Dept. of Computer Science, Cornell University, Ithaca NY, 1979.

225. Desfray P. *Ingénerie des objets: Approche classe-relation application a C++*. Paris: Editions Masson, 1992.

226. Devlin K. *Logic and Information*. Cambridge: University Press, 1991.

227. Diaper D. *CSCW: Psychology, sociology... and computing*. Computer Bulletin, February 1993, p. 22–25.

228. Diaper D. and Sanger C. (Eds.) *CSCW in Practice: An introduction and case studies*. London: Springer-Verlag, 1993.

229. Dick K. and Swett A. *Objectivity/DB*. Object Magazine 5(5), 1995, p. 82–84.

230. Dietrich W.C., Nackman L.R. and Gracer F. *Saving a Legacy with Objects*. In Meyrowitz, 1989.

231. Dillon T. and Tan P.L. *Object-Oriented Conceptual Modeling*. Sydney NSW: Prentice Hall, 1993.

232. Dorfman L. *Object-Oriented Assembly Language*. Pennsylvania: Windcrest, 1990.

233. Dorfman M. and Thayer R.H. *Standards, Guidelines and Examples on System and Software Requirements Engineering*. Los Alamitos, CA: IEEE Computer Society Press, 1990.

234. Douglass B.P. *Real-Time UML*. Reading MA: Addison-Wesley, 1998.

235. Dreyfus H.L. and Dreyfus S.E. *Mind over Machine: The power of human intuition and expertise in the era of the computer*. Oxford: Basil Blackwell, 1986.

236. Dubois D. and Prade H. *Fuzzy Sets and Systems: Theory and Applications*. Academic Press, 1980.

237. Dubois D., Prade H. and Rossazza J.P. *Vagueness, typicality and uncertainty in class hierarchies*. J. of Intelligent Systems 6, 1991, p. 167–183.

238. Duda R., Hart P. *et al. Development of the PROSPECTOR consultation system for mineral exploration*. SRI report projects 5822 and 6415, Menlo Park CA: SRI International, 1976.

239. Dugundji J. *Topology*. Boston MA: Allyn and Bacon, 1966.

240. Duke R., King P., Rose G. and Smith G. *The Object-Z Specification, Version 1*. Software Verification Research Centre, University of Queensland Technical Report 91–1, 1991.

241. Durham A. *IT Horizons* 1(3), 1992.

242. Durham A. *BETA: The pattern language*. Object Magazine 2(4), 1992, p. 82–83.

243. Durr E. *VDM++: A formal specification language for object-oriented designs. In DeWilde and Vandewalle (Eds.)*. IEEE CompEuro 92 Proceedings, IEEE Press, 1992.

244. Eason K.D. *Tools for participation: How managers and users can influence design*. In Knight K. (Ed) Participation in Systems Development. London: Kogan Page, 1989.

245. Eastlake J.J. *A Structured Approach to Computer Strategy*. Chichester: Ellis Horwood, 1987.

246. Eckel B. *Using C++*. Osborne-McGraw Hill, 1989.

247. Edwards J. *Lessons learned in more than ten years of practical application of the Object-Oriented Paradigm*. Proceedings of CASExpo-Europe, London, 1989.

248. Edwards J.S. *Building Knowledge Based Systems*. London: Pitman, 1991.

249. Ege R., Singh M. and Meyer B. (Eds.) *TOOLS8: Proceedings of the eighth international conference on the Technology of Object-Oriented Languages and Systems*. New York: Prentice Hall, 1993.

250. Ehn P. *Work-oriented Design of Computer Artifacts*. Arbetslivscentrum, Stockholm, 1988.

251. Ehn P., Mollervd B. and Sjogren D. *Playing in reality: a paradigm case*. Scand. J. Inf. Systems 2, 1990, p. 101–120.

252. Ehn P. and Kyng M. *Cardboard computers: mocking up hands-on-the future*. In Greenbaum J. and Kyng M. (Eds.) Design at Work: Co-operative Design of Computer Systems. Hillsdale NJ: Lawrence Erlbaum, 1991.

253. Eliëns A. *An object-oriented approach to distributed problem solving*. In Bramer M.A. and Milne R.W. (Eds.) Research and Development in Expert Systems IX. Cambridge University Press, 1992.

254. Ellis M.A. and Stroustrup B. *The Annotated C++ Reference Manual*. Reading MA: Addison-Wesley, 1990.

255. Elmasri R. and Navathe S.B. *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

256. Elmore P., Shaw G.M. and Zdonik S.B. *The ENCORE object-oriented data model*. Technical Report, Brown University, 1989.

257. Embley D.W., Kurtz B.D. and Woodfield S.N. *Object-Oriented Systems Analysis: A Model-Driven Approach*. Englewood Cliffs: Yourdon Press, 1992.

258. Emmerich W. and Schäfer W. *Dedicated object management system benchmarks for software engineering applications*. Proc. Software Engineering Environments, 130–142, Reading, England, 1993.

259. Englebart D. A *Conceptual Framework for the Augmentation of Man's Intellect*. In Howerton P.W. and Weeks D.C. (Eds.) Vistas in Information Handling. Vol 1. Spartan Books, Washington, 1963.

260. Englebart D. and English W.K. *A Research Centre for Augmenting Human Intellect*. In *Proceedings of the 1968 AFIPS Conference*. AFIPS Press — Montale NJ, 1968, p. 395–410.

261. Englemore D. and Morgan A. (Eds.) *Blackboard Systems*. Wokingham, England: Addison-Wesley, 1988.

262. Ericsson K.A. and Simon H.A. *Protocol Analysis: Verbal Reports as Data*. Bradford Books, 1984.

263. Ersavas T. *Squeak as a development platform*. Proc. OOPSLA '99. Reading MA: Addison-Wesley, 1999.

264. Ewald A. and Roy M. *Why OT is good for system integration*. Object Magazine 2(5), 1993, p. 60–63.

265. Fagan M. *Design and Code Inspections and Process Control in the Development of Programs*. IBM Report IBM-SDD-TR-21-572, 1974.

266. Farhoodi F. *CADDIE: an advanced tool for organizational design and process modelling*. In Software Assistance for Business Re-Engineering. Chichester: Wiley, 1994.

267. Feigenbaum E. McCorduck and Nii H.P. *The Rise of the Expert Company*. London: Macmillan, 1988.

268. Ferber J. *Les Systémes Multi-Agents: Vers une intelligence collective*. Paris: InterEditions, 1995.

269. Fichman R.G. and Kemerer C.F. *Adoption of Software Engineering Process Innovations: The Case of Object-Orientation*. Sloan Management Review, Winter 1993, p. 7–22.

270. Fiedler S.P. *Object-Oriented Unit Testing*. Hewlett-Packard Journal, April 1989, p. 69–74.

271. Firesmith D.G. *Object-Oriented Requirements Analysis and Design: A Software Engineering Approach*. Chichester: Wiley, 1993.

272. Firesmith D.G., Henderson-Sellers B. and Graham I.M. *OPEN Modeling Language Reference Manual*. NY: SIGS Books, Cambridge University Press, 1997.

273. Fishman D.H., et al. *Overview of the IRIS DBMS*. In Kim W. and Lochovsky F.H. (Eds.), 1989.

274. Flanagan D. *Java in a Nutshell*. Bonn: O'Reilly & Associates, 1996.

275. Flores F. *The leaders of the future.* In Denning P.J. and Metcalfe R.M. (Eds.) Beyond Calculation: The next 50 years of computing. New York: Copernicus, 1997.

276. Foley J., Kim W., Kovavavics S. and Murray K. *Defining Interfaces at a High Level of Abstraction.* IEEE Software 6(1), 1989, p. 25–32.

277. Forgy C.L. *The OPS5 User's Manual.* Tech Rep CMU-CS-81-135 Computer Science Dept, Carnegie-Mellon University, 1981.

278. Forsythe R. (Ed.) *Expert Systems: Principles and Case Studies. 2nd Edition.* London: Chapman & Hall, 1989.

279. Forsythe R. (Ed.) *Machine Learning.* London: Chapman & Hall, 1989.

280. Foster I. and Taylor S. *Strand: New Concepts in Parallel Programming.* Englewood Cliffs NJ: Prentice Hall, 1990.

281. Fowler M. *Analysis Patterns.* Harlow, England: Addison-Wesley, 1996.

282. Fowler M. *UML Distilled. 2nd Edition.* Harlow, England: Addison-Wesley, 1997.

283. Fowler M. *Refactoring.* Reading MA: Addison-Wesley, 2000.

284. Fowler M. and Capey A. *The use of object-oriented analysis to define a generic model for health care.* Proceedings of SCOOP Europe, London, 1991.

285. Friedman J. *New Options for Object Databases.* Object Magazine 2(6), 1993.

286. Futatsugi K., Goguen J., Jouannaud J-P. and Meseguer J. *Principles of OBJ2.* Proceedings of the 12th Annual Symposium on Principles of Programming Languages. ACM, New York, 1985, p. 52–66.

287. Gabriel R.P. *Patterns of Software.* Oxford: University Press, 1996.

288. Gaines B. and Shaw M. *Documents as Expert Systems.* In Bramer M. and Milne R. (Eds.) Research and Development in Expert Systems IX. Cambridge University Press, 1992.

289. Galitz W. *Human Factors in Office Automation.* Atlanta, GA: Life Office Management Association, 1981.

290. Gamma E. *Object-Oriented Software Development Based on ET++: Design Patterns, Class Library, Tools* (in German). Berlin: Springer-Verlag, 1992.

291. Gamma E., Helm R., Johnson R. and Vlissedes J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading MA: Addison-Wesley, 1995.

292. Gane C. *Computer-Aided Software Engineering: The Methodologies, the Products and the Future.* Englewood Cliffs NJ: Prentice Hall, 1990.

293. Gardner K., Rush A., Crist M., Konitzer R. and Teegarden B. *Cognitive Patterns.* Cambridge: University Press, 1998.

294. Gause D. and Weinberg G. *Exploring.* New York NY: Dorset House, 1989.

295. Gilb T. and Graham D. *Software Inspection.* Wokingham: Addison-Wesley, 1993.

296. Giles R. *Lukasiewitz Logic and Fuzzy Theory.* Int. J. Man–Machine Studies 8, 1976.

297. Gjessing S. and Nygaard K. (Eds.) *ECOOP'88 European Conference on Object-Oriented Programming.* Lecture Notes in Computer Science 276, Berlin: Springer Verlag, 1988.

298. Glass G. and Schuchert B. *The STL Primer.* Upper Saddle River NJ: Prentice Hall, 1996.

299. Goguen J. and Meseguer J. *EQLog: Equality, Types and Geric Modules for Logic Programming*. In DeGroot D. and Lindstrom G. (Eds.) Logic Programming. Englewood Cliffs NJ: Prentice Hall, 1986.

300. Goguen J. and Meseguer J. *Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics*. In Shriver and Wegner, 1987.

301. Gohil N. *Object-Oriented Database Management, Unpublished Bachelor's thesis*. Imperial College of Science and Technology, 1988.

302. Goldberg A. *Smalltalk-80: The Interactive Programming Environment*. Reading MA: Addison-Wesley, 1984.

303. Goldberg A. and Robson D. *Smalltalk-80: The Language and its Implementation*. Reading MA: Addison-Wesley, 1983.

304. Goldberg A. and Rubin K. *Succeeding with Objects*. Reading MA: Addison-Wesley, 1995.

305. Goldberg D.E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading MA: Addison-Wesley, 1989.

306. Goldfarb C.F. and Prescod P. *The XML Handbook*. Upper Saddle River, NJ: Prentice Hall, 1998.

307. Goodall A. *The year of the agent*. AI Watch 3(8), 1994, p. 1–10.

308. Goodwin M. *User Interfaces in C++ and Object-Oriented Programming*. Cambridge MA: MIT Press, 1989.

309. Gorman K. and Choobineh J. *The Object-Oriented Entity-Relationship Model (OOERM)*. J. Man. Info. Systems 33(9), 1989, p. 41–65.

310. Gosling A., Joy B. and Steele G. *The Java Language Specification*. Reading MA: Addison-Wesley, 1996.

311. Graham I.M. *Expert Systems find an Ideal Setting*. Banking Technology, July 1987, p. 17–24.

312. Graham I.M. *Object-Oriented Methods. 1st Edition*. Wokingham, England: Addison-Wesley, 1991.

313. Graham I.M. *Fuzzy Logic in Commercial Expert System Shells: Results and Prospects*. Fuzzy Sets and Systems 40(3), 1991, p. 451–472.

314. Graham I.M. *Corrigendum to Fuzzy Logic in Commercial expert systems — Results and Prospects*. Fuzzy Sets and Systems 43, 1991, p. 337–338.

315. Graham I.M. *Structured Prototyping for Requirements Specification in Conventional IT and Expert Systems*. Computing and Control Engineering Journal 2(2), 1991, p. 82–89.

316. Graham I.M. *A Method for Integrating Object Technology with Rules*. Proceedings of Advanced Information Systems 92, Oxford, Learned Information, 1992.

317. Graham I.M. *Interoperation: Combining object-oriented applications with conventional IT*. Object Magazine 2(4), 1992, p. 36–37.

318. Graham I.M. *Interoperation: combining objects*. Object Magazine 2(4), 1992, p. 36–37.

319. Graham I.M. *Migration strategies column*. Object Magazine 2(5)–3(5), 1993.

320. Graham I.M. *Migration using SOMA: A Semantically Rich Method of Object-Oriented Analysis*. Journal of Object-Oriented Programming 5(9), February 1993.

321. Graham I.M. *On the Impossibility of Artificial Intelligence.* BCS Expert Systems Newsletter, 1993.

322. Graham I.M. *Object-Oriented Methods. 2nd Edition.* Wokingham: Addison-Wesley, 1994.

323. Graham I.M. *Beyond the Use Case: Combining task analysis and scripts in object-oriented requirements capture and business process re-engineering.* In Magnusson, Meyer, Nerson and Perrot, 1994.

324. Graham I.M. *SOMA: Combining object-oriented analysis with rules and task analysis.* In Carmichael, 1994.

325. Graham I.M. *On the Impossibility of Artificial Intelligence.* BCS Specialist Group in Expert Systems Newsletter, Summer 1994.

326. Graham I.M. *Getting to number one with object technology.* Object Manager, October 1994, p. 13–16.

327. Graham I.M. *Migrating to Object Technology.* Wokingham: Addison-Wesley, 1995.

328. Graham I.M. *Task scripts, use cases and scenarios in object-oriented analysis.* Object-Oriented Systems 3(3), 1996, p. 123–142.

329. Graham I.M. *Requirements Engineering and Rapid Development: An Object-Oriented Approach.* Harlow, England: Addison-Wesley, 1998.

330. Graham I.M. and Jones P.L.K. *A Theory of Fuzzy Frames.* In Moralee D.S. (Ed.) *Research and Development in Expert Systems IV.* Cambridge University Press, 1987.

331. Graham I.M. and Jones P.L.K. *Expert Systems: Knowledge, Uncertainty and Decision.* London: Chapman & Hall, 1988.

332. Graham I.M. and Milne R.M. (Eds.) *Research and Development in Expert Systems X.* Cambridge University Press, 1991.

333. Graham I.M., Bischof J. and Henderson-Sellers B. *Associations considered a bad thin.* J. Object-Oriented Programming, 9(9), 1997.

334. Graham I.M., Henderson-Sellers B. and Yanoussi H. *The OPEN Process Specification.* Harlow, England: Addison-Wesley, 1997.

335. Graham I.M. and O'Callaghan A. Migration Strategies, Tutorial Notes: Object World London, 1997.

336. Grand M. *Patterns in Java — Volume 1.* New York: John Wiley, 1998.

337. Grass J.E. *Design Archaeology for Object-Oriented Redesign in C++.* In Korson et al., 1991.

338. Gray P.M.D. *Logic, Algebra and Databases.* Chichester: Ellis Horwood, 1984.

339. Gray P.D. and Mohamed R. *Smalltalk-80: A Practical Introduction.* London: Pitman, 1990.

340. Gray P.M.D. and Kemp G.J.L. *An OODB with entity-based persistence: a protein modelling application.* In Addis T.R. and Muir R.M. (Eds.) *Research and Development in Expert Systems VII.* Cambridge University Press, 1990.

341. Gray P.M.D., Krishnarao G.K. and Paton N.W. *Object-Oriented Databases: A Semantic Data Model Approach.* Englewood Cliffs NJ: Prentice Hall, 1992.

342. Greenberg S. *Computer-supported Cooperative Work and Groupware.* New York: Academic Press, 1991.

343. Gregory R. *Xanadu: Hypertext from the Future.* Dr. Dobbs Journal 75, 1983, p. 28–35.

344. Grossmann R. *Phenomenology and Existentialism: An Introduction.* London: Routledge & Kegan Paul, 1984.

345. Grotehen T. Notes on a meeting with Grady Booch 1995.07.27 (unpublished communication), 1995.

346. Guilfoyle C. and Warner E. *Intelligent Agents: The New Revolution in Software.* Ovum Ltd, 1994.

347. Gupta R. and Horowitz E. *Object-Oriented Databases with Applications to CASE, Networks and VSLI CAD.* Englewood Cliffs NJ: Prentice Hall, 1991.

348. Guttag J., Horning J.J. and Wing J.M. *Larch in Five Easy Pieces.* Palo Alto CA: Digital Systems Research Centre, 1985.

349. Haack S. *Philosophy of Logics.* Cambridge: University Press, 1978.

350. Halasz F.G. *Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems.* Comm. of the ACM, 31(7), 1988, p. 836–852.

351. Halliday S. and Weibel M. *Object-Oriented Software Engineering.* Englewood Cliffs NJ: Prentice Hall, 1993.

352. Halmos P. *Measure Theory.* New York: Van Nostrand, 1950.

353. Halstead M.H. *Elements of Software Science.* Amsterdam: North-Holland, 1977.

354. Hammer M. Reengineering *Work: Don't Automate, Obliterate.* Harvard Business Review. July–August 1990, p. 104–112.

355. Hammer M. and Champy J. *Re-engineering the Corporation: A manifesto for the business revolution.* NY: Harper Collins, 1993.

356. Hammer M. and McLeod D. *The semantic data model: a modelling mechanism for database applications.* Proc. ACM SIGMOD, 1978.

357. Hammer M. and McLeod D. *Database description with SDM: a semantic database model.* ACM Trans. on Database Systems 6, 1981, p. 351–386.

358. Hansen T.L. *The C++ Answer Book.* Reading MA: Addison-Wesley, 1990.

359. Harel D. *Statecharts: A Visual Formalism for Complex Systems.* In Science of Computer Programming 8, 231–274, North Holland, 1987.

360. Harland D.M. *Rekursiv: Object-Oriented Architecture.* Chichester: Wiley, 1988.

361. Harland D.M. and Drummond B. *REKURSIV — Object-Oriented Hardware.* In Blair et al., 1991.

362. Harmon P. and Taylor D.A. *Objects in Action: Commercial applications of object-oriented.* Reading MA: Addison-Wesley, 1993.

363. Harrison W. and Ossher H. *Subject-oriented programming.* Proc. OOPSLA'93, 411–28, Reading MA: ACM Press, 1993.

364. Hart A. *Knowledge Acquisition for Expert Systems. 2nd Edition.* London: Kogan Page, 1989.

365. Hayes I. (Ed.) *Specification Case Studies.* Englewood Cliffs NJ: Prentice Hall, 1987.

366. Hayes P.J. *The Logic of Frames.* In Brachman and Levesque, 1985.

367. Hayes-Roth F., Waterman, D.A. and Lenat D.B. *Building Expert Systems*. Reading MA: Addison-Wesley, 1983.

368. Heeg G., Magnusson B. and Meyer B. (Eds.) *TOOLS7: Proceedings of the seventh international conference on the Technology of Object-Oriented Languages and Systems*. New York: Prentice Hall, 1992.

369. Heitz M. *HOOD: A Hierarchical Object-Oriented Design Method*. Proceedings of the 3rd German ADA Users Congress, Munich, 12-1–12-9, 1988.

370. Hekmatpour S. *Experience with Evolutionary Prototyping in a Large Software Project*. ACM Software Engineering Notes 12(1), 1987, p. 38–41.

371. Hempel C.G. *Philosophy of Natural Science*. Englewood Cliffs NJ: Prentice Hall, 1966.

372. Henderson-Sellers B. *A BOOK of Object-Oriented Knowledge*. Sydney, Aus: Prentice Hall, 1992.

373. Henderson-Sellers B. *The economics of reusing library classes*. JOOP 6(4), 1993, p. 43–50.

374. Henderson-Sellers B. *Object-Oriented Metrics: Measures of Complexity*. Sydney: Prentice Hall, 1996.

375. Henderson-Sellers B. *OPEN relationships — associations, mappings, dependencies and uses*. Journal of Object-Oriented Programming 10(9), 1998, p. 49–57.

376. Henderson-Sellers B. and Constantine L.L. *Object-Oriented Development and Functional Decomposition*. Journal Of Object-Oriented Programming 3(9), 1991, p. 11–17.

377. Henderson-Sellers B. and Edwards J.M. *The Object-Oriented Systems Life Cycle*. Communications of the ACM 33(9), 1990, p. 143–159.

378. Henderson-Sellers B. and Edwards J. *BOOK TWO of Object-Oriented Knowledge: The working object*. Sydney, Aus: Prentice Hall, 1994.

379. Henderson-Sellers B. and Pant Y.R. *Adopting the reuse mindset throughout the lifecycle*. Object Magazine 3(4), 1993, 73–75.

380. Henderson-Sellers B., Constantine L.L. and Graham I.M. *Coupling and cohesion: towards a valid metrics suite for object-oriented analysis and design*. Object-Oriented Systems 3(3), 1996, p. 143–158.

381. Henderson-Sellers B., Simons A. and Yanoussi H. *The OPEN Toolbox of Techniques*. Harlow, England: Addison-Wesley, 1998.

382. Henderson-Sellers B., and Unhelkar B. *OPEN Modelling with UML*. Harlow, England: Addison-Wesley, 2000.

383. Hewitt C. and de Jong P. *Open Systems*. In Brodie et al., 1984.

384. Hickman F.R., Killen J., Land L. *et al. Analysis for Knowledge-based Systems*. Chichester: Ellis Horwood, 1989.

385. Hill R.D. *Supporting concurrency, communication and synchronization in human computer interaction — The Sassafras UIMS*. ACM Transaction on Graphics 5(3), 1986, p. 179–210.

386. Hill R.D. and Hermman M. *The structure of TUBE — A tool for implementing advanced interfaces*. Proceedings of Eurographics'89, Amsterdam: North-Holland, 1989.

387. Hillside Group `http://www.hillside.net/patterns/`, 15 June, 2000.

388. Hoare C.A.R. *Monitors: an operating system structuring concept.* Comms. ACM 17(10), 1974, p. 549–557.

389. Hodgson R. *Finding, Building and Reusing Objects.* Proceedings of Object Oriented Design, Unicom Seminars, Uxbridge, 1990.

390. Hodgson R. *The X Model: A process model for object-oriented software development.* In Proc. of Toulouse'91 (Toulouse, France), 1991, p. 713–728.

391. Hollowell G. *Handbook of Object-Oriented Standards: The object model.* Reading MA: Addison-Wesley, 1993.

392. Holsapple C.W. and Whinston A.B. *Manager's Guide to Expert Systems using Guru.* Dow Jones–Irwin, 1986.

393. HOOD Working Group. *HOOD Reference Manual. Issue 3.0.* European Space Agency, Noordwijk, Netherlands, 1989.

394. Hood R., Kennedy K. and Muller H. *Efficient Recompilation of Module Interfaces in a Software Development Environment.* In Henderson P. (Ed.) Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. ACM Press, 1987, p. 180–189.

395. Hook J. *Understanding Russell — A First Attempt.* In Goos G. and Hartmani J. (Eds.) Semantics of Data Types. Berlin: Springer-Verlag, 1984.

396. Horty J.F., Thomason R.H. and Touretzky D.S. *A Skeptical Theory of Inheritance in Nonmonotonic Semantic Networks.* Artificial Intelligence 42, 1990, p. 311–348.

397. Hu D. *C/C++ Expert Systems.* Cambridge MA: MIT Press, 1989.

398. Huhns M.N. *Distributed Artificial Intelligence.* London: Pitman, 1987.

399. Hull R. and King R. *Semantic Database Modelling: Survey, Applications and Research Issues.* ACM Computing Surveys 19(3), 1987, p. 201–260.

400. Humphrey W.S. *Managing the Software Process.* Reading MA: Addison-Wesley, 1989.

401. Hutchison D. and Walpole J. *Distributed Systems and Objects.* In Blair et al., 1991.

402. IBM. *Common User Access: Guide to User Interface Design.* Cary, NC: International Business Machines, 1991.

403. IBM. *Common User Access: Advanced Interface Design Reference.* Cary, NC: International Business Machines, 1991.

404. IBM. *Object-Oriented Interface Design: CUA guidelines.* New York: QUE, 1992.

405. IEL. *Crystal User Manual.* Richmond, England: Intelligent Environments Limited, 1986.

406. Ilvari J. *Object-Oriented Information Systems Analysis: A framework for object identification.* Trans. of the IEEE, 1991, p. 205–218.

407. Ince D. *Object-Oriented Software Engineering with C++.* London: McGraw-Hill, 1991.

408. Ingalls D.H.H. *The Smalltalk-76: Programming System: Design and Implementation.* Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, 1978.

409. Ishikawa Y. and Tokoro M. *Orient84/K: An Object-Oriented Concurrent Programming Language for Knowledge Representation.* In Yonezawa and Tokoro, 1987, p. 159–198.

410. Jackson M.A. *System Development*. Chichester, England: Prentice Hall, 1983.

411. Jackson M.A. *Software Requirements and Specifications*. Harlow, England: Addison-Wesley, 1995.

412. Jackson M.A. *A Discipline of Description*. Requirements Engineering 3(2), 1998, p. 73–78.

413. Jackson M.A. *Problem Frames and Methods*. Harlow, England: Addison-Wesley, 2001.

414. Jackson P. *Introduction to Expert Systems*. Wokingham: Addison-Wesley, 1986.

415. Jacobs S. *What is Business Process Automation?*. Expert Systems Applications, August 1992, 5–10.

416. Jacobson I., Booch G. and Rumbaugh J. *The Unified Software Development Process*. Reading MA: Addison-Wesley, 1999.

417. Jacobson I., Christerson M., Jonsson P. and Overgaard G. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham: Addison-Wesley, 1992.

418. Jacobson I., Ericsson M. and Jacobson A. *The Object Advantage: Business Process Reengineering with Object Technology*. Wokingham, England: Addison-Wesley, 1995.

419. Jagannathan D. et al. *SIM: A database system based on the semantic data model*. Proc. ACM SIGMOD 88 Conf., 1988, p. 46–55.

420. Jeffcoate J., Hales K. and Downes V. *Object-Oriented Systems: The Commercial Benefits*. Ovum Ltd, London, ISBN 0-903969-42-4, 1989.

421. Jeffries R.E., Anderson A. and Hendrickson C. *Extreme Programming Installed*. Reading MA: Addison-Wesley, 2000.

422. Johnson P. *Human Computer Interaction: Psychology, Task Analysis and software engineering*. London: McGraw-Hill, 1992.

423. Johnson R.E. and Foote B. *Designing Reusable Classes*. Journal Of Object-Oriented Programming 1(2), 1988.

424. Jones C. *Systematic Software Development using VDM*. Englewood Cliffs NJ: Prentice Hall, 1986.

425. Jones C. *Assessment and Control of Software Risks*. Englewood Cliffs NJ: Yourdon Press, 1994.

426. Jones C.B. *Software Development: A rigorous approach*. Englewood Cliffs, NJ: Prentice Hall, 1980.

427. Jones P.L.K. *Practical Rapid Development*. Harlow, England: Addison-Wesley, 1999.

428. Jones T.C. *Programming Productivity*. New York: McGraw-Hill, 1986.

429. Jones T.C. *A short history of function points and feature points*. ACI Computer Services, 1988.

430. Jungclaus R. *Modeling of Dynamic Object Systems*. Weisbaden: Vieweg, 1993.

431. Kaehler T. and Krasner G. *LOOM: Large Object-Oriented Memory for Smalltalk-80 Systems*. In Krasner G. (Ed.) Smalltalk-80: Bits of History, Words of Advice. Reading MA: Addison-Wesley, 1983.

432. Kandel A. *Fuzzy Mathematical Techniques with Applications*. Reading MA: Addison-Wesley, 1986.

433. Kapor M. A *Software Design Manifesto: Time for a Change*. Dr. Dobb's Journal 172, January, 1991, p. 62–68.

434. Kappel G. *Using an object-oriented diagram technique for the design of information systems*. In H.G. Sol and K.M. van Hee (Eds.) Dynamic Modelling of Information Systems. Amsterdam: Elsevier, 1991.

435. Kay A. and Goldberg A. *Personal Dynamic Media*. IEEE Computer 1977 — originally a 1976 Xerox technical report, 1977.

436. Keen P.G.W. and Knapp E.M. *Business Process investment: Getting the right process right*. Harvard: Business School Press, 1995.

437. Keene S. *Object-Oriented Programming in Common Lisp*. Reading MA: Addison-Wesley, 1989.

438. Kelly G.A. *The Psychology of Personal Constructs*. New York: W.W. Norton, 1955.

439. Kendall E.A., Malkoun M.T. and Chong J. *The application of object-oriented analysis to agent-based systems*. J. of Object-Oriented Programming, 9(9), 1997, p. 56–65.

440. Khan N.A. and Jain R. *Explaining Uncertainty in a Distributed Expert System*. In Kowalik J.S. (Ed.) Coupling Symbolic and Numerical Computing in Expert Systems. Amsterdam: North Holland, 1986.

441. Khoshafian S. and Abnous R. *Object Orientation: Concepts, Languages, Databases, User Interfaces*. New York: Wiley, 1990.

442. Khoshafian S., Baker A.B., Abnous R. and Shepherd K. *Object-Oriented Multi-Media Information Management in Client/Server Architectures*. New York: Wiley, 1992.

443. Kiczales G. *Why are black boxes so hard to reuse?*. Notes from OOPSLA'94, 1994.

444. Kiczales G., des Rivières J. and Bobrow D.G. *The Art of the Metaobject Protocol*. Boston MA: MIT Press, 1991.

445. Kiczales G. and Lopes C. *Aspect-Oriented Programming with AspectJ*. Proc. OOPSLA '99, Reading MA: Addison-Wesley, 1999.

446. Kilov H. and Ross J. *Information Modeling: An object-oriented approach*. Englewood Cliffs NJ: Prentice Hall, 1994.

447. Kim W. *Introduction to Object-Oriented Databases*. Cambridge MA: MIT Press, 1990.

448. Kim W. and Lochovsky F.H. (Eds.) *Object-oriented Concepts, Databases and Applications*. Reading MA: Addison-Wesley, 1989.

449. Kim W., Ballou N., Chou H.T. et al. *Features of the ORION object-oriented database*. In Kim and Lochovsky (Eds.), 1989.

450. King R. *My Cat is Object-Oriented*. In Kim and Lochovsky, 1989.

451. Kirkerud B. *Object-Oriented Programming with SIMULA*. Wokingham: Addison-Wesley, 1989.

452. Kitagawa H. and Kunii T. *The unnormalized relational data model for office form processor design*. Berlin: Springer-Verlag, 1989.

453. Kleppe A. and Warmer J. *Making UML activity diagrams object-oriented*. Proc. Tools Europe 2000, Upper Saddle River NJ: Prentice Hall, 2000.

454. Knudsen J.L., Lofgren M., Lehrmann-Madsen O. and Magnusson B. (Eds.) *Object-Oriented Environments: The Mjolner approach*. Hemel Hempstead, England: Prentice Hall, 1994.

455. Korson T. and McGregor J.D. *Technical Criteria for the Specification and Evaluation of object-oriented Libraries*. Proceedings of Object Expo Europe 1993, New York: SIGS Publications, 1993, p. 33–36.

456. Korson T., Vaishnavi V. and Meyer B. (Eds.) *TOOLS5: Proc. Fifth International Conference on the Technology of Object-Oriented Languages and Systems*. New York: Prentice Hall, 1991.

457. Kosko B. *Neural Networks and Fuzzy Systems: A dynamical systems approach to machine intelligence*. Englewood Cliffs, NJ: Prentice Hall, 1991.

458. Kosko B. *Fuzzy Thinking: The new science of fuzzy logic*. New York: Hyperion, 1993.

459. Koza J.R. *Genetic Programming: On the programming of computers by means of natural selection*. Cambridge MA: MIT Press, 1992.

460. Kristen G. *Object-Orientation: The KISS Method*. Wokingham, England: Addison-Wesley, 1995.

461. Kristensen B.B., Madsen O.L., Moller-Pedersen B. and Nygaard K. *The BETA Programming Language*. In Shriver and Wegner, 1987, p. 7–48.

462. Kruchten P. *The 4+1 View of Software Architecture*. IEEE Software November 1995, p. 42–50.

463. Kruchten P. *The Rational Unified Process*. Reading MA: Addison-Wesley, 1999.

464. Kurtz B., Woodfield S. and Embley D. *Object-Oriented Systems Analysis and Specification: A Model Driven Approach*. Englewood Cliffs NJ: Prentice Hall, 1991.

465. LaLonde W.R. and Pugh J.R. *Inside Smalltalk: Volume I.*, Englewood Cliffs NJ: Prentice Hall, 1990.

466. LaLonde W.R. and Pugh J.R. *Inside Smalltalk: Volume II*. Englewood Cliffs NJ: Prentice Hall, 1991.

467. Lang K. and Perlmutter B. *Oaklisp: An object-oriented Scheme with First Class Types*. In Meyrowitz, 1986, p. 30–37.

468. Lano K. and Haughton H. *Reasoning and Refinement in object-oriented specification languages*. Proc. of ECOOP'92, Berlin: Springer, 1992.

469. Lano K. and Haughton H. (Eds.) *Object-Oriented Specification Case Studies*. Englewood Cliffs, NJ: Prentice Hall, 1994.

470. Laranjeira L.A. *Software Size Estimation of Object-Oriented Systems*. IEEE Transactions on Software Engineering 16(5), 1990, p. 510–522.

471. Larman C. *Applying UML and Patterns*. Upper Saddle River NJ: Prentice Hall, 1998.

472. Laurel B. (Ed.) *The Art of Computer Interface Design*. Reading, MA: Addison-Wesley, 1990.

473. Lausen G. and Vossen G. *Models and Languages of Object-Oriented Databases*. Harlow, England: Addison-Wesley, 1998.

474. Lea D. *Christopher Alexander: An Introduction for object-oriented designers*. ACM SIGSOFT 19(1), 1994, p. 39–46.

475. Leathers B. *Cognos and Eiffel: A Cautionary Tale.* Hotline on Object-Oriented Technology 1(9), 1990, p. 1–8.

476. Lécluse C., Richard P. and Velez F. *O2, an object-oriented data model.* In Zdonik and Maier, 1990.

477. Lee G. *Object-Oriented GUI Application Development.* Englewood Cliffs, NJ: Prentice Hall, 1993.

478. Lee S. and Carver D.L. *Object-Oriented Analysis and Specification: A knowledge based approach.* Journal Of Object-Oriented Programming 3(9), 1991, p. 35–43.

479. Lieberherr K., Holland I., Lee G. and Riel A. *Object-oriented programming: an objective sense of style.* IEEE Computer, 21(6), 1988.

480. Leffingwell D. and Widrig D. *Managing Software Requirements: a unified approach.* Reading MA: Addison-Wesley, 2000.

481. Lektorsky V.A. *Subject, Object, Cognition.* Moscow: Progress Publishers, 1984.

482. Lemay L. and Perkins C.L. *Teach Yourself Java in 21 days. 2nd Edition.* Indiana: Sams.net, 1997.

483. Lenat D.B. and Guha R.V. *Building Large Knowledge Based Systems: Representation and Inference in the Cyc Project.* Reading MA: Addison-Wesley, 1990.

484. Lenin V.I. *The State and Revolution.* Collected Works Vol 25, Moscow: Progress Publishers, 1964.

485. Lenzerini M., Nardi D. and Simi M. (Eds.) *Inheritance Hierarchies in Knowledge Representation and Programming Languages.* Chichester, England: Wiley, 1991.

486. Lewis E. (Ed.) *Object-Oriented Application Frameworks.* Greenwich, CT: Manning, 1995.

487. Lientz B.P. and Swanson E.B. *Software Maintenance: A User/Management Tug of War.* Data Management, April 1979, p. 26–30.

488. Linnemann V. *Non First Normal Form Relations and Recursive Queries: An SQL-Based Approach.* IEEE Conference on Data Engineering, 1987, p. 591–598.

489. Lippman S.B. and Lajoie J. *C++ Primer. 3rd Edition.* Reading: Addison-Wesley, 1998.

490. Liskov B., Snyder A., Atkinson R. and Schaffert. *Abstraction Mechanisms in CLU.* Communications of the ACM, 20(8), 1977.

491. Liu C. *Smalltalk, Objects and Design.* Greenwich MA: Manning, 1996.

492. Lloyd D. *LISP is no Impediment!.* Systems International, January 1990, p. 43–45.

493. Londeix B. *Cost Estimation for Software Development.* Wokingham: Addison-Wesley, 1987.

494. Loosely C. *Separation and integration in the Zachman framework.* Database Newsletter 20(1), 1992, p. 3–9.

495. Lorenz M. *Object-Oriented Software Development: A Practical Guide.* Englewood Cliffs NJ: Prentice Hall, 1993.

496. Lorenz M. and Kidd J. *Object-Oriented Software Metrics.* Englewood Cliffs NJ: Prentice Hall, 1994.

497. Love T. *Object Lessons.* New York: SIGS Books, 1992.

498. Lu and Zhou. *Selective Inheritance in Fuzzy Object Modelling*. Unpublished manuscript, 2000.

499. Macaulay L.A. *Requirements Engineering*. London: Springer, 1996.

500. Machiavelli N. *The Prince*. Translation by G. Bull. Harmondsworth: Penguin Books, 1961.

501. Mackenzie K.D. *The Organizational Hologram: The effective management of organizational change*. Boston: Kluwer Academic, 1991.

502. MacLane S. *Categories for the Working Mathematician*. New York: Springer, 1971.

503. MacLean R., Stepney S., Smith S., Tordoff N., Gradwell D. and Hoverd T. *Analysing Systems: Determining Requirements for Object-Oriented Development*. Hemel Hempstead, England: Prentice Hall, 1994.

504. MacLennan B.J. *Values and Objects in Programming Languages*. SIGPLAN Notices 17(12), 1982, p. 70–79.

505. MacQueen D. *Using dependent types to express modular structure*. In Proc. of 13th Annual ACM Symposium on Principles of Programming Languages (St. Petersburg Beach, Fla, Jan) New York: ACM, 1986, p. 277–286.

506. Madsen O.L. and Moller-Pedersen B. *What object-oriented programming may be — and what it does not have to be*. In Gjessing and Nygaard, 1988.

507. Madsen O.L., Moller-Pedersen B. and Nygaard K. *Object-Oriented Programming in the BETA Programming Language*. Wokingham, England: Addison-Wesley, 1993.

508. Magnusson B., Meyer B., Nerson J-M. and Perrot J-F. (Eds.) *TOOLS 13*. Hemel Hempstead, England: Prentice Hall, 1994.

509. Maiden N.A.M. and Rugg G. *ACRE: selecting methods for requirements acquisition*. IEE Software Engineering Journal, May 1996, p. 183–192.

510. Maiden N.A.M. and Sutcliffe A.G. *Requirements critiquing using domain abstractions*. Proc. of IEEE Conference on Requirements Engineering, IEEE Computer Society Press, 1994, p. 184–193.

511. Maiden N.A.M., Cisse M., Perez,H. and Manuel D. *CREWS Validation Frames: Patterns for Validating Systems Requirements*. Centre for Human Computer Interface Design, City University, London, 1998.

512. Maiden N.A.M., Minocha S., Manning K. and Ryan M. *SAVRE: Systematic Scenario Generation and Use*. Centre for HCI Design, City University, London, 1997.

513. Maier D. and Stein J. *Development and Implementation of an Object-Oriented DBMS*. In Shriver and Wegner, 1987.

514. Mak V.W. *Connection: an inter-component communication paradigm for configurable distributed systems*. In Proc. of Int. Workshop on Configurable Distributed Systems, London, 1992.

515. Malan R., Letsinger R. and Coleman D. *Object-Oriented Development at Work*. Upper Saddle River NJ: Prentice Hall, 1996.

516. Mandrioli D. and Meyer B. *Advances in Object-Oriented Software Engineering*. Englewood Cliffs NJ: Prentice Hall, 1992.

517. Manola F. and Dayal U. PDM: *An Object-Oriented Data Model.* International Workshop on Object-Oriented Database Systems, California, 1986.

518. Marcuse H. *Reason and Revolution: Hegel and the Rise of Social Theory.* London: Oxford University Press, 1941.

519. Marcuse H. *Reason and Revolution.* Oxford: University Press, 1955.

520. Marden R.J. *Object-Oriented System Development: Notation and Method* (unpublished manuscript), 1990.

521. Martin J. and Odell J.J. *Object-Oriented Analysis And Design.* Englewood Cliffs NJ: Prentice Hall, 1992.

522. Martin J. and Odell J.J. *Object-Oriented Methods: A Foundation.* Englewood Cliffs NJ: Prentice Hall, 1995.

523. Martin J. and Odell J.J. *Object-Oriented Methods: Pragmatic Considerations.* Englewood Cliffs NJ: Prentice Hall, 1996.

524. Martin J. and Odell J.J. *Object-Oriented Methods: A Foundation (UML Edition).* Englewood Cliffs NJ: Prentice Hall, 1998.

525. Martin-Löf P. *An Intuitionistic Theory of Types: Predicative Part.* In Rose H.E. and Shepherdson J.C. (Eds.) Logic Colloquium 1973. Amsterdam: North-Holland, 1975, p. 73–118.

526. Martin-Löf P. *Constructive Mathematics and Computer Programming.* In *Logic, Methodology and Philosophy of Science VI* (Proceedings of the 6th International Congress, Hanover 1979). Amsterdam: North-Holland, 1982, p. 153–175.

527. Marx G. *Computerworld,* 20 April, 1992.

528. Marx K. *Capital, Afterword to the second German edition, translated by Moore and Aveling.* Moscow: Foreign Languages Publishing House, vol. I, 1961.

529. Masiero P. and Germano F.S.R. *JSD as an Object-Oriented Design Method.* Software Engineering Notes 13(3), 1988, p. 22–23.

530. Matthews D. *Programming Language Design with Polymorphism.* PhD Dissertation, Computer Lab., University of Cambridge, England, 1983.

531. McCabe F.G. *Logic and Objects.* Englewood Cliffs NJ: Prentice Hall, 1992.

532. McCabe T.J. A *complexity measure.* IEEE Trans. on Software Engineering, 2(4), 1976, p. 308–320.

533. McCarthy J. *Recursive Functions of Symbolic Expressions and their Computation by Machine. Part I.* Comm. of the ACM 3(4), 1960.

534. McCauley C. *CORBA at Irish Life.* Proc. Enterprise CORBA, London: ELM Ltd, 1999.

535. McCawley J.D. *Everything that Linguists have always wanted to know about Logic (but were ashamed to ask).* Oxford: Basil Blackwell, 1981.

536. McDermott D. and Doyle J. Nonmonotonic Logic I, Artificial Intelligence 13(1,2), 1980.

537. McGraw G. *Securing Java.* New York: Wiley, 1998.

538. McGregor J.D. and Sykes D.A. *Object-Oriented Software Development: Engineering Software for Reuse.* New York: Van Nostrand, 1992.

539. McInnes S.T. *The Generic Relational Model.* PhD Thesis, Department of Computer Science, University of Strathclyde, Glasgow, 1988.

540. McLarty C. *Elementary Categories, Elementary Toposes.* Oxford University Press, 1992.

541. Meersman R.A., Kent W. and Khosla S. (Eds.) *Object-Oriented Databases: Analysis, Design and Construction (DS-4).* Proceedings of the IFIP TC2/WG2.6 Working Conference, Windermere, England, July 1991, Berlin, Springer-Verlag.

542. Meira S. and Cavalcanti A. *Modular object-oriented Z specifications.* In Z Users Meeting 1990. Workshops in Computing, Berlin: Springer, 1992.

543. Menzies T., Edwards J.M. and Ng K. *The case of the mysterious missing reusable libraries.* In Meyer B. and Potter J. (Eds.) TOOLS 9. Sydney: Prentice Hall, 1992.

544. Meyer B. *Object-oriented Software Construction.* Englewood Cliffs NJ: Prentice Hal, 1988.

545. Meyer B. *Eiffel: The Language and the Environment.* Englewood Cliffs NJ: Prentice Hall, 1990.

546. Meyer B. (Ed.) *Special Issue: Concurrent Object-Oriented Programming.* Comms. ACM, 36(9), 1993.

547. Meyer B. *Reusable Software: The Base Object-Oriented Component Libraries.* Hemel Hempstead, England: Prentice Hall, 1994.

548. Meyer B. *Object Success.* Hemel Hempstead, England: Prentice Hall, 1995.

549. Meyer B. *Object-oriented Software Construction. 2nd Edition.* Upper Saddle River NJ: Prentice Hall, 1997.

550. Meyer B. and Mandrioli D. (Eds.) *Advances in Object-Oriented Software Engineering.* Englewood Cliffs NJ: Prentice Hall, 1992.

551. Meyer B. and Nerson J-M. (Eds.) *Object-Oriented Applications.* Englewood Cliffs NJ: Prentice Hall, 1993.

552. Meyrowitz N. (Ed.) *OOPSLA'86 ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM SIGPLAN Notices **21** (11), 1986.

553. Meyrowitz N. (Ed.) *OOPSLA'87 ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM SIGPLAN Notices **22** (12), 1987.

554. Meyrowitz N. (Ed.) *OOPSLA'88 ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM SIGPLAN Notices **23** (11), 1988.

555. Meyrowitz N. (Ed.) *OOPSLA'89 ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* Reading MA: Addison-Wesley, 1989.

556. Meyrowitz N. '(Ed.) *OOPSLA'90 ACM Conference on Object-Oriented Programming Systems, Languages and Applications.* Reading, MA: Addison-Wesley, 1990.

557. Michaels W.I. *Re-designing customer driven IS processes.* Database Newsletter 20(6), 1992, p. 1–12.

558. Microsoft *The Windows Interface: An Application Design Guide.* Seattle: Microsoft Corp, 1995.

559. Miller A.V. *Hegel's Science of Logic* (translation). London: George, Allen and Unwin, 1969.

560. Miller G.A. *The magical number seven, plus or minus two: some limits on our capacity for processing informatiom.* Psychological Review 63, 1956, p. 81–97.

561. Milner R. *A Theory of Type Polymorphism in Programming.* J. Comput. Syst. Sci. 17, 1978, p. 348–375.

562. Minsky M.L. *A Framework for Representing Knowledge.* In Haugeland (Ed.). Mind Design. MIT Press, 1981.

563. Minsky M.L. *The Society of Mind.* London: Heinemann, 1985.

564. Minsky M.L. and Papert S. *Perceptrons.* MIT Press, 1969.

565. Mitchell J. and Plotkin G. *Abstract types have existential type.* In Proc. of 12th Annual ACM Symposium on Principles of Programming Languages (New Orleans, LA, Jan) New York: ACM, 1985, p. 37–51.

566. Mock M. *DoubleVision: A Foundation for Scientific Visualization.* In Pinson and Wiener, 1990.

567. Monarchi D.E. and Puhr G.I. *A Research Typology for Object-Oriented Analysis and Design.* Comms, ACM 35(9), 1992, p. 35–47.

568. Moon D. *Object-Oriented Programming with Flavors.* In Meyrowitz, 1986, p. 1–8.

569. Moon D. *The Common Lisp object-oriented programming Language Standard.* In Kim and Lochovsky, 1989.

570. Monday P., Carey J. and Dangler M. *San Francisco Component Framework.* Reading MA: Addison-Wesley, 2000.

571. Moran T.P. *The Command Language Grammar: A representation for the user interface of interactive computer systems.* IJMMS 15, 1981, p. 3–50.

572. Morgan G. *Images of Organization.* Thousands Oaks, CA: Sage, 1997.

573. Mowbray T. and Malveau R. *CORBA Design Patterns.* New York: Wiley, 1997.

574. Mowbray T. and Zahavi R. *The Essential CORBA.* New York: Wiley, 1995.

575. Mullender S. (Ed.) *Distributed Systems.* Reading MA: Addison-Wesley, 1989.

576. Müller H.J. and Dieng R. (Eds.) *Computational Conflicts.* Heidelberg: Springer, 2000.

577. Mullin M. *Object-Oriented Program Design.* Wokingham: Addison-Wesley, 1989.

578. Mumford E. *Designing Systems for Business Success: The ETHICS Method.* Manchester: Business School, 1986.

579. Musser D.R. and Saini A. *STL Tutorial and Refererence Guide.* Reading MA: Addison-Wesley, 1996.

580. Myers B.A. *GARNET.* IEEE Software 8(2), 1991.

581. Myers G.J. *The Art of Software Testing.* New York: Wiley, 1979.

582. Naur P. and Randell B. (Eds.) *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee.* 7–11 October 1968, Garmisch, Germany: NATO Science Committee, 1968.

583. Nelson T.H. *Replacing the Printed Word: A Complete Literary System.* In Lavington S.H. (Ed.) *Information Processing 80.* Amsterdam: North Holland, 1980, p. 1013–1023.

584. Nelson T.H. *Literary Machines.* Nelson, Swathmore PA, 1981.

585. Nerson J. *Applying Object-Oriented Analysis and Design*. Comms. ACM 35(9), 1992, p. 63–74.

586. Newell A. and Simon H.A. GPS: *A Program that Simulates Human Thought*. In Feigenbaum E.A. and Feldman J.A. (Eds.). Computers and Thought. McGraw Hill, 1963.

587. NeXT Computers Inc. *The NeXTSTEP Environment: Concepts*. NeXT Computers Inc., 1991.

588. NeXT Computers Inc. *NeXTSTEP User Interface Guidelines: Release 3*. Reading MA: Addison-Wesley, 1992.

589. Nguyen G.T. and Rieu D. *Schema Evolution in Object-Oriented Database Systems*. Data & Knowledge Engineering 4(1), 1989, p. 43–68.

590. Nguyen Q. and Van Le T. *A fuzzy dynamic multiple inheritance model in object-oriented simulation*. Proc. European Simulation Multiconference, Budapest, Hungary, 1996, p. 624–628.

591. Nierstrasz O. and Tsichritzis D.E. *Integrated Office Systems*. In Kim and Lochovsky, 1989.

592. Nierstrasz O., Gibbs S. and Tsichritzis D.E. *Component-oriented software development*. Comms. ACM 35(9), 1992, p. 160–165.

593. Nordström B., Petersson K. and Smith J.M. *Programming in Martin-Löf's Type Theory*. Oxford: University Press, 1990.

594. Nori K.V., Amman U., Jensen K., Nageli H. and Jacobi C. *Pascal-P implementation notes*. In Barron D. Pascal: The Language and its Implementation. New York: Wiley, 1991.

595. Norman D.A. *The Psychology of Everyday Things*. New York: Doubleday, 1988.

596. Norman D.A. and Draper S.W. *User Centred System Design*. Hillsdale NJ: Lawrence Erlbaum, 1986.

597. O'Callaghan A.J. *Object Technology Skills: What, Why and How*. In O'Callaghan A.J. and Leigh M. (Eds.) Object Technology Transfer. Henley-on-Thames: Alfred Waller, 1994.

598. O'Callaghan A. *Object-oriented reverse engineering*. Application Development Adviser 1(1), 1997, p. 35–39.

599. O'Callaghan A. *Realizing the reality*. Application Development Adviser 1(2), 1997, p. 30–33.

600. O'Callaghan A. *A plethora of patterns*. Application Development Adviser 1(3), 1998, p. 32–33.

601. O'Callaghan A.J. *Migrating Large-Scale Legacy Systems to Component-based and Object Technology: The Evolution of a Pattern Language*. Communications of the AIS, 2(3), 1999 (`http://cais.isworld.org/`).

602. O'Callaghan A.J. *Getting what you want*. Application Development Advisor 3(5), 2000, p. 64–67.

603. O'Callaghan A.J. *Patterns for an Architectural Praxis*. Proc. European Pattern Languages of Program Design, Irsee, Germany, 2000.

604. Object Management Group. *Common Object Request Broker Architecture*. Framington MA: Object Management Group, 1992.

605. O'Connor P. *A knowledge based trading system that management can trust*. Expert Systems User, July 1991.

606. O'Sullivan J. *XML Messaging at Chase Manhattan Bank Global Markets*. Proc. OT'99, Oxford, England, 1999.

607. Odell J.J. *Six different kinds of composition*. J. of Object-Oriented Programming 5(8), 1994, p. 10–15.

608. ODI *ObjectStore Release 2.0 User Guide and Reference Manual*. Object Design Inc., 1 New England Executive Park, Burlington MA, 1992.

609. Open Software Foundation. *OSF/Motif: Style Guide*. Englewood Cliffs NJ: Prentice Hall, 1990.

610. Otte R., Patrick P. and Roy M. *Understanding CORBA*. Upper Saddle River NJ: Prentice Hall, 1996.

611. Ousterhout J.K. *Tcl and the Tk Toolkit*. Reading MA: Addison-Wesley, 1994.

612. Ovum Ltd. *Workflow Management Software: The Business Opportunity*. London: Ovum, 1992.

613. Paepcke A. (Ed.) *OOPSLA'91 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*. Reading MA: Addison-Wesley, 1991.

614. Page-Jones M. *Comparing Techniques by means of Encapsulation and Connascence*. Comms. ACM 35(9), 1992, p. 147–151.

615. Page-Jones M., Constantine L.L. and Weiss S. *Modelling object-oriented systems: The Uniform Object Notation*. Computer Language, October 1990, p. 70–87.

616. Page-Jones and Weiss S. *Synthesis: An object-oriented analysis and design method*. American Programmer 2(7), 1989, p. 64–67.

617. Parnas D. *On the Criteria to be Used in Decomposing Systems into Modules*. Comm. ACM 15(2), 1972, p. 1053–1058.

618. Parsaye K., Chignell M., Khoshafian S. and Wong H. *Intelligent Databases: Object-Oriented, Deductive Hypermedia*. New York: Wiley, 1989.

619. Parson J. and Wand Y. *Using objects for systems analysis*. Communications of the ACM, 40(12), 1995, p. 104–110.

620. Peckham J. and Maryanski J. *Semantic Data Models*. ACM Computing Surveys 20(3), 1988, p. 153–189.

621. Pedrycz and Sosnowski *Fuzzy object-oriented system design*. Fuzzy Sets and Systems 99(2), 1998, p. 121–134.

622. Perry D.E. and Kaiser G.E. *Adequate testing and object-oriented programming*. Journal Of Object-Oriented Programming 3(5), 1990, p. 13–19.

623. Perry D.E and Wolf A.L. *Foundations for the Study of Software Architecture*. Software Engineering Notes 17(4), 1992, p. 40–52.

624. Peters T. *Thriving on Chaos: Handbook for a Management Revoloution*. New York: Alfred Knopf, 1987.

625. Peters T. *Liberation Management: Necessary Disorganization for the Nanosecond Nineties*. London: Macmillan, 1992.

626. Peters T.J. and Waterman R.H. *In Search of Excellence*. New York: Harper & Row, 1982.

627. Peterson J.L. *Petri Net Theory and the Modelling of Systems*. Englewood Cliffs, NJ: Prentice Hall, 1981.

628. Petri C.A. *Kommunikation mit Automaten*. Ph.D dissertation, University of Bonn, 1962.

629. Peuquet D.J. and Marble D.F. (Eds.) *Introductory readings in Geographic Information Systems*. London: Taylor and Francis, 1990.

630. Pinson L.J. and Weiner R.S. *An Introduction to Object-Oriented Programming and C++*. Reading MA: Addison-Wesley, 1988.

631. Pinson L.J. and Weiner R.S. *Object-Oriented Design of a Branch Path Analyzer for C-Language Software Systems*. In Pinson and Wiener, 1990.

632. Pinson L.J. and Weiner R.S. (Eds.) *Applications of Object-Oriented Programming*. Reading MA: Addison-Wesley, 1990.

633. Pohl I. *C++ for C Programmers. 2nd Edition*. Benjamin/Cummings, 1994.

634. Pohl I. *C++ Distilled*. Reading MA: Addison-Wesley, 1996.

635. Pohl K. *The three dimensions of requirements engineering*. In Rolland C., Bodart F. and Cauvet C. (Eds) Proc. CAISE'93. Paris: Springer, 1993, p. 175–292.

636. Pohl K., Jarke M. and Weidenhaupt K. The use of scenarios during systems development — the current state of practice, submitted to *ICRE*, 1997.

637. Porter M. *Competitive Strategy: Techniques for analyzing industries and competitors*. New York: The Free Press, 1980.

638. Potter J. and Takoro M. (Eds.) *TOOLS6: Proceedings of the sixth international conference on the Technology of Object-Oriented Languages and Systems*. New York: Prentice Hall, 1992.

639. Pountain D. *Rekursiv: An object-oriented CPU*. Byte 13(11), 1988, p. 341–349.

640. Pountain D. *Occam II*. Byte 14(10), 1989, p. 279–84.

641. Pree W. *Design Patterns for Object-Oriented Software Development*. Reading MA: Addison-Wesley, 1995.

642. Prieto-Diaz R. *Status Report: Software Reusability*. IEEE Software, May 1993, p. 61–66.

643. Prieto-Diaz R. and Freeman P. *Classifying Software for Reuseability*. IEEE Software 4(1), 1987, p. 6–16.

644. Putnam L.H. *A General Empirical Solution to the Macro Software Sizing and Estimating Problem*. IEEE Trans. on Software Engineering, July 1978, p. 345–361.

645. Quillian M.R. *Word Concepts: A Theory and Simulation of some Basic Semantic Capabilities*. Behavioural Science 12, 1967, p. 410–430.

646. Quinlan J.R. *Discovering Rules from Large Collections of Examples: A case study*. In Michie D. (Ed.) Expert Systems in the Microelectronics Age. Edinburgh University Press, 1979.

647. Raghavan R. *Building Interactive Graphical Applications Using C++*. In Pinson and Weiner, 1990.

648. Rational. *UML Notation Guide*. Internet: `www.rational.com`, 1997.

649. Rawlings R. *Some Numbers from an Object-Oriented Development*. Object-Oriented Software Engineering, BCS ITEXT, 1991.

650. Reason J.T. *Human Error.* Cambridge: University Press, 1990.

651. Redmond-Pyle D. and Graham I.M. *Object-Oriented Methods In Europe, paper presented at Data Management'92.* Bristol University, 1992.

652. Redmond-Pyle D. and Moore A. *Graphical User Interface Design and Evaluation.* London: Prentice Hall, 1995.

653. Reenskaug T.M.H. *User-oriented Descriptions of Smalltalk Systems.* Byte, August 1981, p. 149–166.

654. Reenskaug T., Wold P. and Lehne A. *Working with Objects: The OOram Software Engineering Method.* Englewood Cliffs NJ: Prentice Hall, 1996.

655. Reisig W. *Petri Nets: An introduction.* Berlin: Springer, 1986.

656. Reisner P. *Formal Grammars and Human Factors Design of an Interactive Graphics System.* IEEE Trans. on Software Engineering 5, 1981, p. 229–240.

657. Reiss S.P. *Tools for Object-Oriented Redesign.* In Korson et al., 1991.

658. Reiter R. *On Reasoning by Default.* In Brachman and Levesque, 1985.

659. Rentsch T. *Object-oriented Programming.* SIGPLAN Notices 17(9), 1982, p. 51–79.

660. Riecken D. *Special Issue on Intelligent Agents.* Comms ACM, July 1994.

661. Rising L. (Ed.) *The Patterns Handbook.* New York: Cambridge University Press, 1998.

662. Roach S.S. *Services under siege: The restructuring imperative.* Harvard Business-Review, September-October 1991, p. 82–92.

663. Roberts R.B. and Goldstein I.P. The FRL manual, AI Memo. No. 409, MIT Artificial Intelligence Laboratory, 1977.

664. Robinson J.A. *A machine-oriented logic based on the resolution principle.* Proc. of the ACM 12, 1965.

665. Robinson K. and Berrisford G. *Object-Oriented SSADM.* Englewood Cliffs NJ: Prentice Hall, 1994.

666. Robinson P. (Ed.) *Object-Oriented Design.* London: Chapman & Hall, 1992.

667. Rosenberg J. and Koch D. (Eds.) *Persistent Object Systems.* Berlin: Springer-Verlag, 1990.

668. Rosene F. *A Software Development Environment called STEP.* In Proc. ACM Conference on Software Tools, 1995.

669. Ross J. and Kilov H. *Information Modelling: An object-oriented approach.* Englewood Cliffs, NJ: Prentice Hall, 1994.

670. Royce W. *Software Project Management: A Unified Framework.* Reading MA: Addison-Wesley, 1998.

671. Royce W.W. *Managing the development of large scale software system.* Proc. IEEE WESCON, 1970, p. 1–9.

672. Rubin K. and Goldberg A. *Object Behaviour Analysis.* Comms. of the ACM 35(9), September 1992, p. 48–62.

673. Rumbaugh J. *Forceful Functions: How to do numerical computation.* Object Magazine 6(6), 1993, p. 18–24.

674. Rumbaugh J., Blaha M., Premerlani W. et al. *Object-Oriented Modelling and Design.* Englewood Cliffs NJ: Prentice Hall, 1991.

675. Rumbaugh J., Booch G. and Jacobson I. *The Unified Modelling Language Reference Manual.* Reading MA: Addison-Wesley, 1999.

676. Rumbaugh J. and Selic B. *Using UML for Modeling Complex Real-time Systems*, 15 June 2000 (`http://www.objectime.com/otl/technical/umlrt_overview.pdf`).

677. Rummelhart D.E. and McClelland J.L. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* Cambridge MA: MIT Press, 1986.

678. Rush G. *The fast way to define system requirements.* Computerworld, 7 October, 1985.

679. Russell S. and Norvig P. *Artificial Intelligence: A Modern Approach.* Englewood Cliffs NJ: Prentice Hall, 1995.

680. Saeki M., Horai H. and Enomoto H. *Software Development Process from Natural Language Specification.* Proceedings of the 11th International Conference on Software Engineering, New York: IEEE Computer Society, 1989.

681. Sakkinen M. *Comments on 'the Law of Demeter' and C++.* SIGPLAN Notices 23(12), 1988, p. 38.

682. Schaffert C., Cooper T., Bullis B., Kilian M. and Wilpot C. *An Introduction to Trellis/Owl.* In Meyrowitz, 1986, p. 9–16.

683. Schank R.C. and Abelson R.P. *Scripts, Plans, Goals and Understanding.* Lawrence Erlbaum Associates, 1977.

684. Scharenberg M.E. and Dunsmore H.E. *Evolution of classes and objects during object-oriented design and programming.* Journal Of Object-Oriented Programming 3(9), 1991, p. 30–34.

685. Schlageter G., Unland R., et al. *OOPS: An Object-Oriented Programming System with Integrated Data Management Facility.* Proc. IEEE 4th Int. Conf. on Data Engineering, Los Angeles, 1988, p. 118–125.

686. Schmid H.A. and Swenson J.R. *On the Semantics of the Relational Data Model.* Proc. 1975 ACM SIGMOD International Conference on Management of Data, 1975.

687. Schmucker K. *Object-Oriented Programming for the Macintosh.* Hayden Book Company, 1986.

688. Schneiderman B. *Designing the User Interface: Strategies for effective human computer interaction.* Reading MA: Addison-Wesley, 1987.

689. Schreiber G., Wielinger R. and Breuker J. (Eds.) *KADS: A Principled Approach to Knowledge Based System Development.* London: Academic Press, 1993.

690. Scott Gordon V. and Bieman J.M. *Reported effects of rapid prototyping on industrial software quality.* Software Quality Journal 2, 1993, p. 93–108.

691. Scott A.C., Clayton J.E. and Gibson E.L. *A Practical Approach to Knowledge Acquisition.* Reading MA: Addison-Wesley, 1991.

692. Searle J. R. *Speech Acts.* Cambridge: University Press, 1969.

693. Seidewitz E. and Stark M. *General Object-oriented Software Development.* Software Engineering Letters 86–002, 1986.

694. Selic B., Gullekson G. and Ward P.T. *Real-time Object-Oriented Modelling*. New York: Wiley, 1994.

695. Senge P.M. *The Fifth Discipline: The Art And Practice Of The Learning Organization*. New York: Doubleday; London: Random House, 1990.

696. Servio Logic. *Programming in OPAL*. Beaverton OR: Servio Logic Devmt. Corp, 1989.

697. Shadbolt N. *Knowledge Representation in Man and Machine*. In Forsythe R. Expert Systems: Principles and Case Studies. 2nd Edition. Chapman & Hall, 1989.

698. Shafer D. and Taylor D.A. *Transforming the Enterprise through cooperation: An object-oriented solution*. Englewood Cliffs, NJ: Prentice Hall, 1993.

699. Sharble R.S. and Cohen S. *The object-oriented brewery: a comparison of two OO development methods*. ACM Sigsoft 18(2), 1994.

700. Shastri L. *Semantic Networks: An Evidential Formalization and its Connectionist Realization*. London: Pitman, 1988.

701. Shaw M. *Prospects for an Engineering Discipline of Software*. IEEE Software 7(6), 1990, p. 15–24.

702. Shaw M., DeLine R., Klein D.V., Ross T.L., Young D.M. and Zelesnik G. *Abstractions for Software Architecture and Tools to Support Them*. IEEE Trans. on Software Eng., 21(4), 1995, p. 314–315.

703. Shaw M. and Gaines B. *On the Relationship between Repertory Grid and Term Subsumption Knowledge Structures: Theory, Practice and Tools*. In Bramer M. and Milne R. (Eds.) Research and Development in Expert Systems IX. Cambridge University Press, 1992.

704. Shaw M. and Garlan D. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs NJ: Prentice Hall, 1996.

705. Shelton R.E. *An Object-Oriented Method for Enterprise Modeling: OOEM*. Proceedings of OOP'93, Munich, 61–70, New York: SIGS Publications, 1993.

706. Shipman D. *The Functional Data Model and the Data Language DAPLEX*. ACM TODS 6(1), 1981.

707. Shlaer S. and Mellor S.J. *Object-Oriented Systems Analysis — Modelling the World in Data*. Englewood Cliffs NJ: Yourdon Press, 1988.

708. Shlaer S. and Mellor S.J. *Object-Lifecycles: Modelling the World in States*. Yourdon Press, 1992.

709. Shneiderman B. *User Interface Design for the Hyperties Electronic Encylopedia*. Proceedings of the Hypertext '87 Workshop, University of North Carolina at Chapel Hill, 1987.

710. Short J.E. and Venkatramen N. *Beyond business process redesign: Redefining Baxter's Business Network*. Sloan Management Review, Fall 1992, p. 7–17.

711. Shortliffe E.H. *Computer Based Medical Consultations: MYCIN*. American Elsevier, 1976.

712. Shriver B. and Wegner P. (Eds.) *Research Directions in Object-oriented Programming*. Cambridge MA: MIT Press, 1987.

713. Simon H.A. *The Shape of Automation for Men and Managemen* (publisher unknown), 1965.

714. Simon H.A. *The Sciences of the Artificial. 2nd Edition*. Cambridge MA: MIT Press, 1981.

715. Simons A. and Graham I. *30 things that go wrong in object modelling with UML 1.3*. In Kilov, Rumpe and Simmonds (Eds.). Behavioral Specifications of Businesses and Systems. Kluwer Academic Publishers, 1999.

716. Sims O. *Business Objects: Delivering Cooperative Objects for Client-Server*. London: McGraw-Hill, 1994.

717. Skarra A.H. and Zdonik S.B. *The Management of Changing Types in an Object-Oriented Database*. In Meyrowitz, 1986.

718. Smith J.M. and Smith D.C.P. *Database abstractions — aggregation and generalization*. ACM Trans. on Databaser Systems 2, 1977, p. 105–133.

719. Smith M.D. and Robson D.J. *A framework for testing object-oriented programs*. JOOP 5(3), 1992, p. 45–53.

720. Snyder A. *Inheritance and the Development of Encapsulated Software Systems*. In Shriver and Wegner, 1987.

721. Software Futures. *Objects and Software Reuse: Dead on arrival at your company*. Software Futures 3(3), 1994.

722. Soley R.M. (Ed.) *Object Management Architecture Guide*. Framington MA: Object Management Group, 1990.

723. Sommerville I. *Software Engineering. 3rd Edition*. Wokingham: Addison-Wesley, 1989.

724. Sowa J.F. *Conceptual Structures: Information processing in mind and machine*. Reading MA: Addison-Wesley, 1984.

725. Sowa J.F. and Zachman J.A. *Extending and formalizing the framework for information systems architecture*. IBM Systems Journal 31(3), 1992, p. 590–616.

726. Stapleton J. *Dynamic Systems Development Method: The Method in Practice*. Harlow, England: Addison-Wesley, 1997.

727. Stephenson N. *Snow Crash*. Harmondsworth: Penguin Books. 1992.

728. Stern R. *Hegel, Kant and the Structure of the Object*. London: Routledge, 1990.

729. Stoecklin S.E., Adams E.J. and Smith S. *Object-Oriented Analysis*. Proceedings of the 5th Washington ADA Symposium, ACM, New York, 1988, p. 133–138.

730. Stonebraker M.R. and Rowe L.A. (Eds.) *The Postgres Papers*. Research memo. UCB/ERL M86/85, University of California, Berkeley, 1987.

731. Stonebraker M.R., Rowe L.A., Lindsay B., Gray P., Carey Brodie M.L., Bernstein P. and Beech D. *The Third Generation Database System Manifesto*. Proceedings of the 1990 SIGMOD Conference, ACM, 1990.

732. Stout L. *A survey of fuzzy set and topos theory*. Fuzzy Sets and Systems 42(1), 1991, p. 3–14.

733. Stroustrup B. *The C++ Programming Language*. Reading MA: Addison-Wesley, 1986.

734. Stroustrup B. *What is Object-Oriented Programming?*. IEEE Software, May 1988, p. 10–20.

735. Stroustrup B. *The Design and Evolution of C++*. Reading MA: Addison-Wesley, 1994.

736. Stroustrup B. *The C++ Programming Language. 3rd Edition.* Reading MA: Addison-Wesley, 1997.

737. Stroustrup B. Interview in *Application Development Adviser*, 2000.

738. Suchman L.A. *Plans and Situated Actions: The problem of human-machine communication.* Cambridge, Cambridge University Press, 1987.

739. Sully P. *Modelling the World with Objects.* Englewood Cliffs NJ: Prentice Hall, 1993.

740. Sun Microsystems Inc. *Open Look: Graphical User Interface Functional Specification.* Reading MA: Addison-Wesley, 1989.

741. Swaffield G.E. *Development of a Structured Method for Knowledge Elicitation.* PhD Thesis, Thames Polytechnic, London, 1990.

742. Swatman P.A. and Swatman P.M.C. *Formal specification: an analytic tool for (management) information systems.* J. of Information Systems, 2(2), 1992, p. 121–160.

743. Symbolics. *Statice.* Cambridge MA: Symbolics Inc., 1988.

744. Symons G. *Software Sizing and Estimating: Mk II Function Point Analysis.* Chichester: Wiley, 1989.

745. Szyperski C. *Component Software: Beyond Object-Oriented Programming.* Harlow, England: Addison-Wesley, 1998.

746. Szyperski C., Omohundro S. and Murer S. *Engineering a programming language — the type and class system of Sather.* Lecture Notes in Computer Science 782, Berlin: Springer, 1994.

747. Tanenbaum A.S. and Renesse R. van. *Distributed Operating Systems.* ACM Computing Surveys 17(4), 1985, p. 419–470.

748. Taylor D. *Object-Oriented Information Systems: Planning and Implementation.* New York: Wiley, 1992.

749. Taylor D. *Object-Oriented Technology: A Manager's Guide.* Reading MA: Addison-Wesley, 1992.

750. Taylor D. *Object-Oriented Technology: A Manager's Guide. 2nd Edition.* Reading MA: Addison-Wesley, 1997.

751. Taylor D.A. *Business Engineering with Object Technology.* New York: John Wiley & Sons, 1995.

752. Tello E.R. *Object-oriented Programming for Artificial Intelligence: A Guide to Tools and System Design.* Reading MA: Addison-Wesley, 1989.

753. Ten Dyke R.P. and Kunz J.C. *Object-Oriented Programming.* IBM Systems Journal 28(3), 1989.

754. Teorey T.J., Yang D.Q. and Fry J.P. *A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model.* ACM Computing Surveys 18(2), 1986, 197–222.

755. Tesler L. *The Smalltalk Environment.* BYTE August 1981, p. 90–147.

756. Texel P.T. and Williams C.B. *Use Cases Combined with Booch/OMT/UML: Process and Products.* Upper Saddle River NJ: Prentice Hall, 1997.

757. Thimbleby H. *User Interface Design.* New York: ACM Press (Addison-Wesley), 1990.

758. Tognazzini B. *TOG on Interface*. Reading MA: Addison-Wesley, 1992.

759. Tomlinson C. and Scheevel M. *Concurrent Object-Oriented Programming Languages*. In Kim and Lochovsky, 1989.

760. Topper A. *Object-Oriented Development in COBOL*. New York: McGraw-Hill, 1995.

761. Touretzky D.S. *The Mathematics of Inheritance Systems*. London: Pitman, 1986.

762. Turner D.A. *Miranda: A Non-Strict Functional Language with Polymorphic Types*. In Jouannaud J-P. (Ed.) Functional Programming Languages and Computer Architectures. Berlin: Springer Lecture Notes in Computer Science 201, 1985, p. 1–16.

763. Turner R. *Logics for Artificial Intelligence*. Chichester: Ellis Horwood, 1984.

764. Ullman J.D. *Principles of Database Systems*. Maryland: Computer Science Press, 1981.

765. Ullman J.D. *Principles of Database and Knowledge-base Systems*. Vol. I. Computer Science Press, 1989.

766. Ullman J.D. *Principles of Database and Knowledge-base Systems*. Vol. II — The New Technologies, Computer Science Press, 1989.

767. Ungar D. and Smith R.B. *Self: The power of simplicity*. In Meyrowitz, 1987.

768. Van Gyseghem N. and De Caluwe R. *Overview of the UFO database model*. Proc. 4th European congress on Intelligent Techniques and Soft Computing (EUEIT), Aachen, Germany, 1996, p. 858–862.

769. Van Harmelen M. *Object Modelling and User Interface Design*. Harlow, England: Addison-Wesley, 2000.

770. Van Rijsbergen C.J. *The State of Information Retrieval: Logic and information*. Computer Bulletin, February 1993, p. 18–20.

771. Vlissides J., Kerth N. and Coplien J.O. (Eds.) *Pattern Languages of Program Design 2*. Reading MA: Addison-Wesley, 1996.

772. Waldén K. and Nerson J-M. *Seamless Object-Oriented Software Architecture*. NY: Prentice Hall, 1995.

773. Wand Y. *A Proposal for a Formal Model of Objects*. In Kim W. and Lochovsky F.H. (Eds.) Object-oriented Concepts, Databases and Applications. Reading MA: Addison-Wesley, 1989.

774. Ward P. *How to Integrate Object-Orientation with Structured Analysis and Design*. IEEE Software 6, March 1989.

775. Warmer J. and Kleppe A. *The Object Constraint Language*. Reading MA: Addison-Wesley, 1999.

776. Wasserman A.I., Pircher P.A. and Muller R.J. *The Object-oriented Structured Design Notation for Software Design Representation*. IEEE Computer, March 1990, p. 50–62.

777. Webster J. *Shaping Women's Work: Gender, Employment and Information Technology*. London: Longman Sociology, 1996.

778. Wegner P. *The object-oriented classification paradigm*. In Shriver B. and Wegner P. (Eds.) Research Directions in Object-Oriented Programming. Cambridge: University Press, 1987.

779. Wegner P. and Zdonik S. *Inheritance as an Incremental Modification Mechanism*. Proc. ECOOP'88, Lecture Notes in Computer Science 322, 1988, p. 55–77, New York: Springer.

780. Weiner N. *Cybernetics*. Cambridge MA: MIT Press, 1948.

781. Weiner R. and Pinson L. *The C++ Workbook*. Reading MA: Addison-Wesley, 1990.

782. Weiser S.P. and Lochovsky F.H. *OZ+: An object-oriented database system*. In Kim and Lochovsky, 1989.

783. Weiskamp K. and Fleming B. *The Complete C++ Primer*. Academic Press, 1990.

784. Wellman F. *Software Costing: An objective approach to estimating and controlling the cost of computer software*. London: Prentice Hall, 1992.

785. Weyuker E. *Evaluating software complexity measures*. IEEE Trans. on Software Engineering 14(9), 1988, p. 1357–1365.

786. Whitewater Group. *Actor Language Manual*. Evanston IL: The Whitewater Group Inc., 1989.

787. Whitmire S.A. *Object-Oriented Design Measurement*. New York: Wiley, 1997.

788. Wieringa R.J. *Requirements Engineering*. Chichester, England: Wiley, 1996.

789. Wilkie F.G. *Object-Oriented Software Engineering: A Guide from Concepts to Practice*. Wokingham: Addison-Wesley, 1993.

790. Wills A.C. *Specification in Fresco*. In Stepney, Harden and Cooper (Eds.) Object-Orientation in Z. Berlin: Springer, 1991.

791. Wills A.C. *Frameworks and CBD*. In Patel D., Sun Y. and Patel S. (Eds.) Proc. OO Info. Systems. Berlin: Springer, 1996.

792. Wills A.C. *Understanding and Using Catalysis*. Harlow, England: Addison-Wesley (provisional title), 2001.

793. Wilson D. *Class diagrams: A tool for design, documentation and teaching*. Journal of Object-Oriented Programming 3(9), 1990, p. 38–44.

794. Wilson P. *Computer Supported Cooperative Work*. Oxford: Intellect Books, 1990.

795. Winblad A.L., Edwards S.D. and King D.R. *Object-Oriented Software*. Reading MA: Addison-Wesley, 1990.

796. Winder R. *Developing C++ Software. 2nd Edition*. Chichester: Wiley, 1993.

797. Winder R. *Developing Java Software. 2nd Edition*. Chichester: Wiley, 2000.

798. Wing J.M. and Nixon M.R. (1989) *Extending Ina Jo with Temporal Logic*. IEEE Transactions on Software Engineering, February 1989.

799. Winograd T. (Ed.) *Bringing Design to Software*. Reading MA: Addison-Wesley, 1996.

800. Winograd T. and Flores F. *Understanding Computers and Cognition*. Reading MA: Addison-Wesley, 1986.

801. Winston P.H. *Artificial Intelligence. 2nd Edition*. Reading MA: Addison-Wesley, 1984.

802. Wirfs-Brock R. and McKean A. *Responsibility-Driven Design* (unpublished tutorial notes), 1996.

803. Wirfs-Brock R., Wilkerson B. and Wiener L. *Designing Object-Oriented Software*. Englewood Cliffs NJ: Prentice Hall, 1990.

804. Wirth N. *Algorithms + Data = Programs*. Englewood Cliffs, NJ: Prentice Hall, 1976.

805. Witt B.I., Baker F.T. and Merritt E.W. *Software Architecture and Design — Principles, Models and Methods*. New York: Van Nostrand Rheinhold, 1994.

806. Woods W.A. *What's in a Link*. In Brachman and Levesque, 1985.

807. Woolfe R. *Managing and redesigning business processes to achieve dramatic performance improvements*. European Business Journal, 1991.

808. WWISA (World Wide Institute of Software Architects). `www.wwisa.org`. 14 June, 2000.

809. Wu C.T. *Development of a Visual Database Interface: An Object-Oriented Approach*. In Pinson and Weiner, 1990.

810. Wulf W.A., London R.L. and Shaw M. *An Introduction to the Construction and Verification of Alphard Programs*. IEEE Trans. on Software Engineering, SE-2, 1976.

811. Xephon. *The Mainframe Market Monitor*. Newbury, England: Xephon, 1992.

812. Yazici A., George R., Buckles B.P. et al. *A survey of conceptual and logical data models for uncertainty management*. In Zadeh L. and Kacprzyk J. (Eds.) Fuzzy Logic for the Management of Uncertainty. New York: Wiley, 1992.

813. Yokote Y. and Tokoro M. *Concurrent Programming in ConcurrentSmalltalk*. In Yonezawa and Tokoro, 1987, p. 129–158.

814. Yonezawa A. and Tokoro M. (Eds.) *Object-Oriented Concurrent Programming*. Cambidge MA: MIT Press, 1987.

815. Young J.Z. *Philosophy and the Brain*. Oxford University Press, 1986.

816. Young R.M., Green T.R.G. and Simon T. *Programmable User Models for Predictive Evaluation of Interface Designs*. In Bice K. and Lewis C. (Eds.) Human Factors in Computer Systems — CHI'89. Reading MA: Addison-Wesley, 1989.

817. Yourdon E. *The Decline and Fall of the American Programmer*. Englewood Cliffs NJ: Prentice Hall, 1992.

818. Yourdon E. *Object-Oriented Systems Design*. Englewood Cliffs NJ: Prentice Hall, 1994.

819. Yourdon E. and Constantine L.L. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood Cliffs NJ: Prentice Hall, 1979.

820. Yourdon E., Whitehead K., Thomann J., Oppel K. and Nevermann P. *Mainstream Objects: An Analysis and Design Approach for Business*. Englewood Cliffs NJ: Prentice Hall, 1995.

821. Yuan G. *A depth-first process model for object-oriented development with improved OOA/OOD notations*. Report on Object-Oriented Analysis and Design 2(1), 1995, p. 23–37.

822. Zachman J.A. *A framework for information systems architecture*. IBM Systems Journal 26(3), 1987, p. 276–292.

823. Zadeh L.A. *Fuzzy Sets*. Information and Control 8, 1965.

824. Zadeh L.A. *Similarity relations and fuzzy orderings*. Information Sciences 3, 1971, p. 177–200.

825. Zadeh L.A. *Outline of a New Approach to the Analysis of Complex Systems and Decision Processes*. IEEE Trans. Systems Man. and Cybernetics, SMC-3, 1973, p. 28–44.

826. Zadeh L.A. A *Computational Approach to Fuzzy Quantifiers in Natural Languages*. University of California (Berkeley) Memo. UCB-ERL M82-36, 1982.

827. Zadeh L.A. *Syllogistic Reasoning in Fuzzy Logic and its Application to Usuality and Reasoning with Dispositions*. IEEE Transactions SMC-15(6), 1985.

828. Zamir S. *Handbook of Object Technology*. Boca Raton FL: CRC Press, 1999.

829. Zdonik S.B. and Maier D. (Eds.) *Readings in Object-Oriented Database Systems*. Morgan Kaufmann, 1990.

830. Zdonik S.B. and Wegner P. *A Database Approach to Languages, Libraries and Environments*. Technical Report CS-85-10, Department of Computer Science, Brown University, 1985.

831. Zdonik S.B. and Wegner P. *Language and Methodology for Object-Oriented Database Environments*. Technical Report CS-85-19, Department of Computer Science, Brown University, 1985.

Предметный указатель

D

E

T

U

V

W

X

A

Б

В

Г

К

Л

М

Н

О

Р

С

Ф

Х

Ц

Ч

Ш

Э

Алфавитный указатель авторов

Научно-популярное издание

Иан Грэхем

Объектно-ориентированные методы. Принципы и практика
3-е издание

Литературный редактор *Е.Д. Давидян*
Верстка *М.А. Смолина*
Художественные редакторы *Г.В. Базылев, В.Г. Павлютин*
Корректоры *З.В. Александрова, Л.А. Гордиенко, Л.В. Коровкина, О.В. Мишутина, Л.В. Чернокозинская*

Издательский дом "Вильямс".
101509, Москва, ул. Лесная, д 43, стр 1
Изд. лиц. ЛР № 090230 от 18.02 04
Госкомитета РФ по печати

Подписано в печать 18 02.2004 Формат 70×100/16
Гарнитура NewtonC. Печать офсетная
Усл печ л 70,95 Уч.-изд л 62,33
Тираж 2500 экз Заказ № 1976

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций
197110 Санкт-Петербург, Чкаловский пр , 15

Объектно-ориентированные методы

Принципы и практика

Третье издание

Иан Грэхем

Авторитетное и полное руководство по объектным технологиям, не привязанное к конкретному языку программирования

Со времени выхода в свет предыдущего издания этой книги в области объектных технологий произошли колоссальные изменения. Поэтому материал книги был полностью переосмыслен и модифицирован в целях отражения новейших технологий и методологий, включая принципы создания программного обеспечения среднего уровня, компонентную разработку, языки Java и UML. Если вы разработчик или руководитель проекта, в котором планируется применять объектные технологии, эта книга обеспечит полное понимание ключевых концепций, преимуществ и недостатков этого подхода. Более того, она поможет определиться с выбором средств реализации проекта. Книга содержит результаты философского осмысления и описание современного опыта применения объектно-ориентированных технологий и продуктов.

Основные особенности этого издания

- Детальное описание программного обеспечения среднего уровня и стратегий перехода на новую платформу
- Описание проверенных на практике наиболее удачных принципов анализа и проектирования с углубленным рассмотрением вопросов архитектуры и шаблонов, а также строгое представление метода Catalysis, предназначенного для разработки приложений на основе компонентов
- Модифицированное описание инженерии требований с подробным рассмотрением подхода SOMA
- Описание языка Java и других объектно-ориентированных языков программирования

Кроме того

- Существенно доработана глава, посвященная объектно-ориентированным базам данных, с учетом появления новых и модифицированных программных продуктов
- Добавлен обзор процессов разработки и методов управления проектами, включая RUP и OPEN Process, а также приведены рекомендации по тестированию и проектированию пользовательского интерфейса
- Проанализированы система обозначений языка UML и около 50 объектно-ориентированных методов
- Ответы на вопросы и упражнения приведены на Web-сайте по адресу: www.trireme.com

Об авторе

Иан Грэхем — всемирно известный консультант в области объектных технологий и моделирования бизнес-процессов. Он обладает более чем двадцатилетним опытом практической разработки информационных технологий и в настоящее время является главным консультантом компании TriReme International Ltd. Иан — автор, редактор и публицист, он принимал участие в написании и издании тринадцати книг. Как талантливого популяризатора знаний его регулярно приглашают на конференции по всему миру.

Более подробную информацию о серии Object Technology можно найти по адресу: http://www.awl.com/cseng/otseries.

ISBN 5-8459-0438-2

03032

9 785845 904386

ВИЛЬЯМС

Издательский дом "ВИЛЬЯМС"
www.williamspublishing.com

Addison-Wesley
Pearson Education