UML ОСНОВЫ

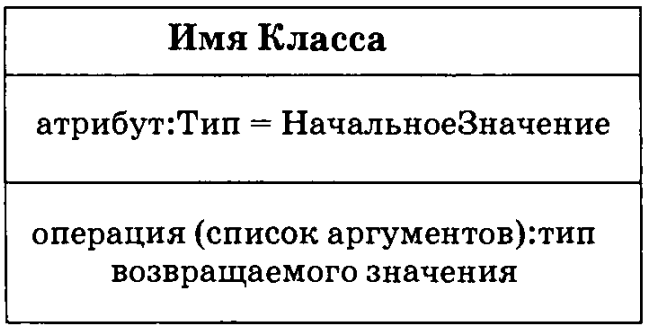
ВТОРОЕ ИЗДАНИЕ

КРАТКОЕ РУКОВОДСТВО ПО УНИФИЦИРОВАННОМУ ЯЗЫКУ МОДЕЛИРОВАНИЯ

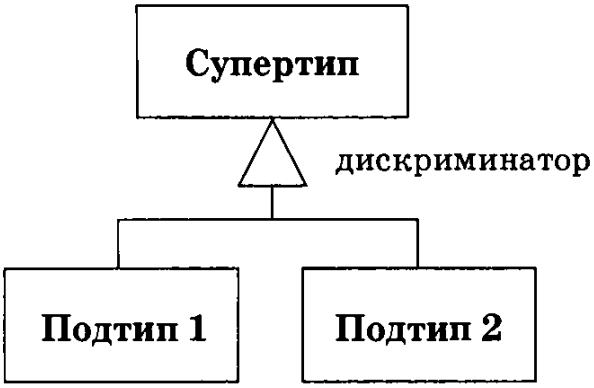
МАРТИН ФАУЛЕР
и КЕНДАЛЛ СКОТТ

Предисловие Гради Буча,
Айвара Джекобсона и Джеймса Рамбо

Класс



Обобщение



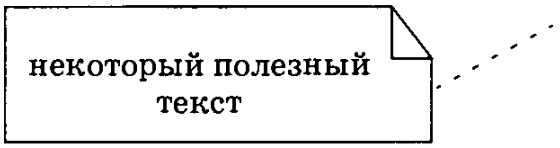
Ограничение

{описание ограничения}

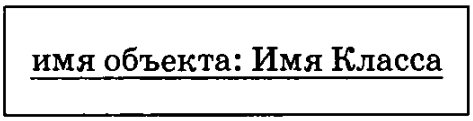
Стереотип

«имя стереотипа»

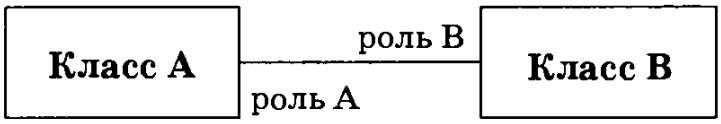
Примечание



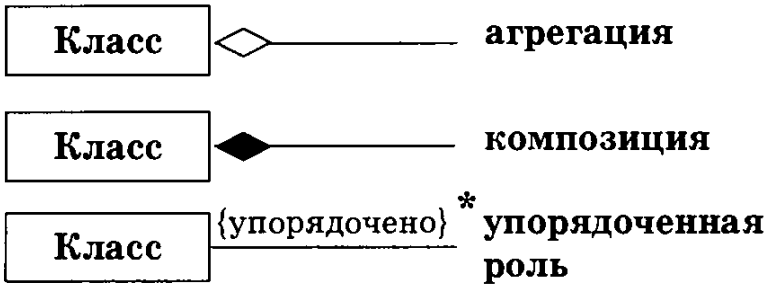
Объект



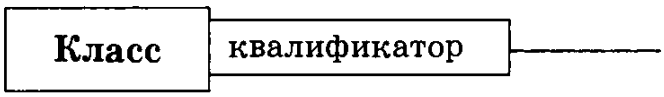
Ассоциация



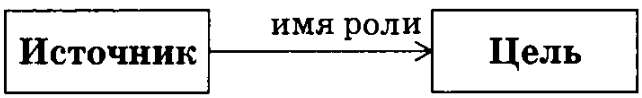
Кратности



Квалифицированная ассоциация



Навигация



Зависимость

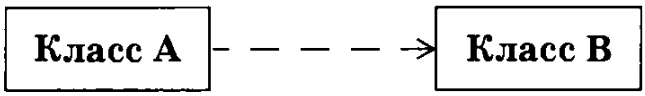


Диаграмма классов: Интерфейсы

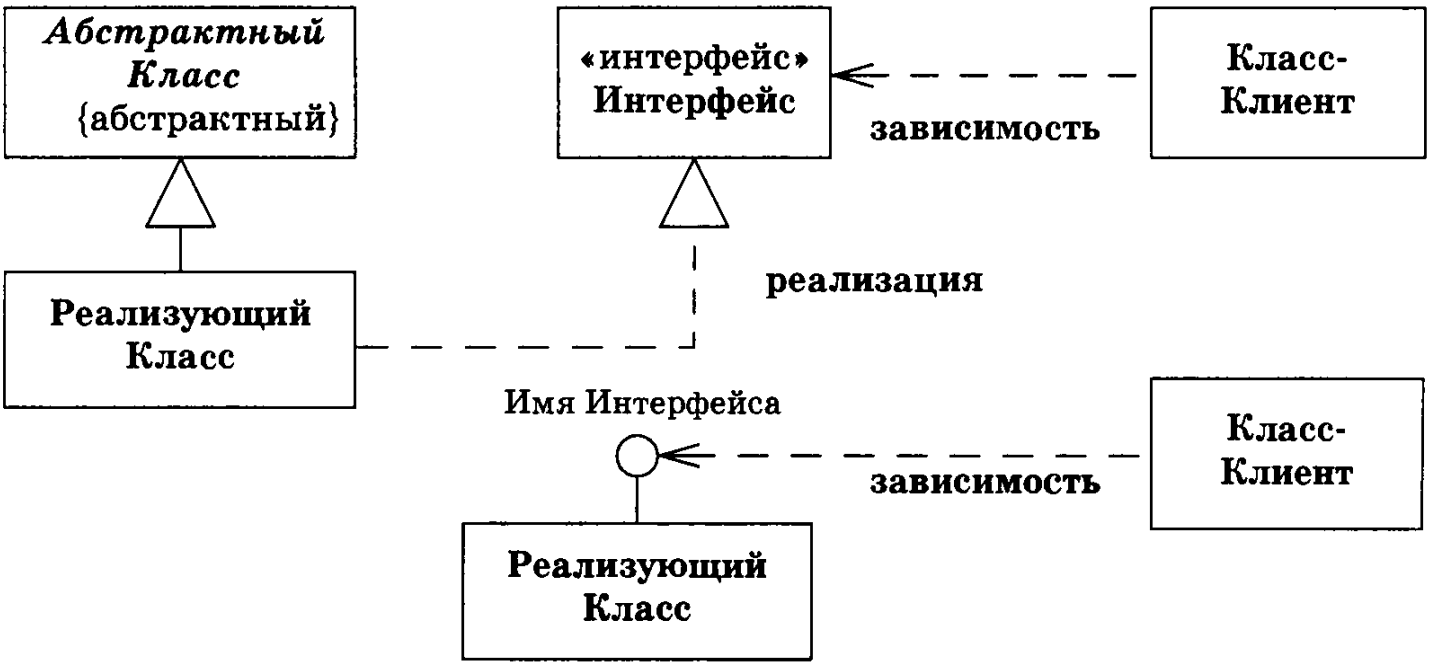


Диаграмма классов: Параметризованный класс

Класс-Ассоциация

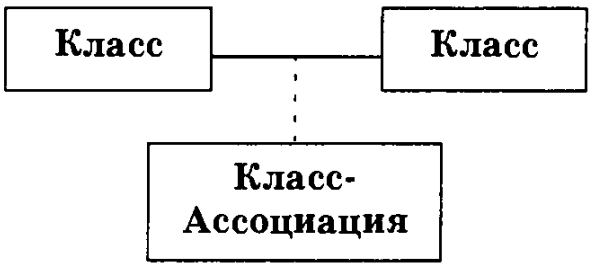
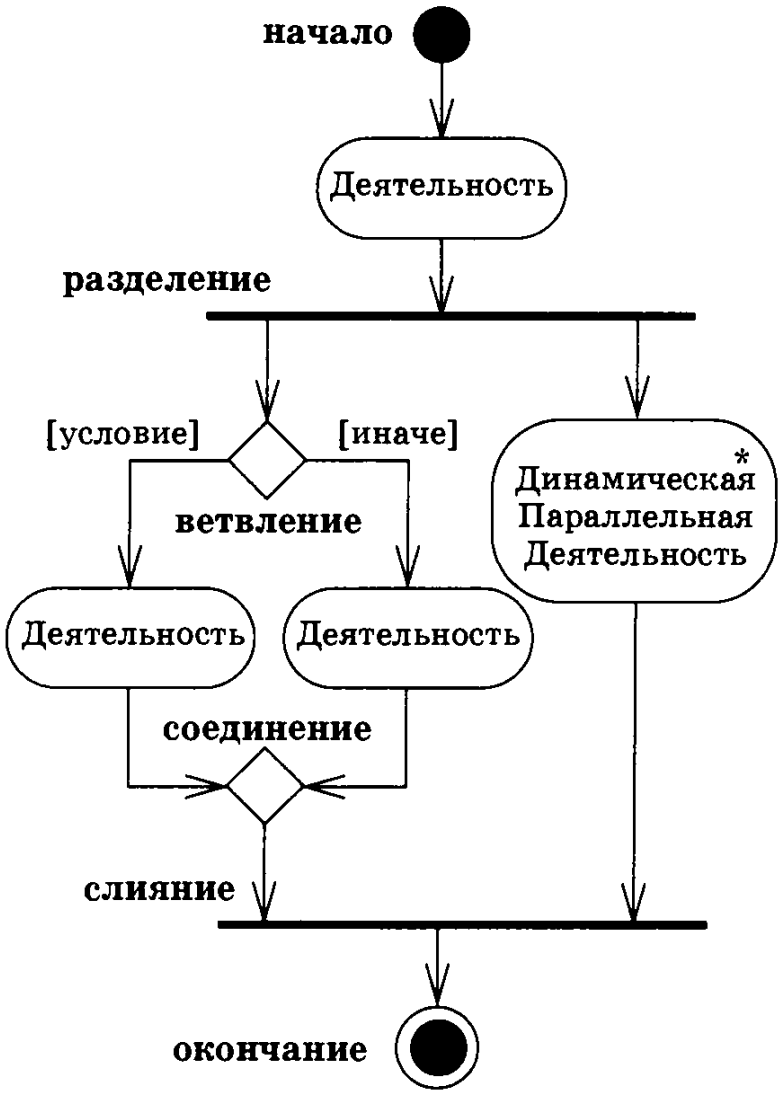


Диаграмма деятельности



Мартин Фаулер и Кендалл Скотт

UML. Основы

Перевод А. Леоненкова

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Редактор	*Н. Макарова*
Корректура	*И. Воробьева*
Верстка	*Н. Гриценко, А. Дорошенко*

Фаулер М., Скотт К.
UML. Основы. – Пер. с англ. – СПб: Символ-Плюс, 2002. – 192 с., ил.
ISBN 5-93286-032-4

Первое издание «UML в кратком изложении» стало бестселлером и получило высокую оценку создателей языка UML, Г. Буча, А. Джекобсона и Д. Рамбо. Язык UML стал стандартным способом изображения диаграмм в объектно-ориентированных проектах. Второе издание, написанное с учетом последних изменений UML, сохранило краткий стиль, что позволит быстро выяснить, какие элементы нотации языка являются ключевыми, что они означают и как их применять. Издание существенно дополнено, в том числе диаграммами вариантов использования, диаграммами деятельности и расширения кооперации, а также новым приложением, описывающим отличия разных версий UML.

Книга написана для тех, кто знаком с основами объектно-ориентированного анализа и проектирования. М. Фаулер рассматривает в контексте UML различные методы моделирования, такие как варианты использования, диаграммы классов и диаграммы взаимодействия, и описывает ясно и кратко нотацию и семантику. Также представлены полезные не-UML методы – CRC-карты и образцы. Книга содержит множество полезных рекомендаций, основанных на 12-летнем опыте автора, и пример UML-проекта, реализованного на Java.

ISBN 5-93286-032-4
ISBN 0-201-65783-X (англ)

Издательство «Символ-Плюс». 193148, Санкт-Петербург, ул. Пинегина, 4,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП № 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 19.11.2001. Формат 70×100$^{1}/_{16}$. Печать офсетная.
Объем 12 печ. л. Тираж 3000 экз. Заказ № 2089.
Отпечатано с диапозитивов в ФГУП «Печатный двор» им. А. М. Горького Министерства РФ
по делам печати, телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Оглавление

Предисловие

Когда мы приступили к созданию унифицированного языка моделирования UML (Unified Modeling Language), то надеялись, что сможем разработать стандартное средство для спецификации проектов, которое будет не только отражать наилучший практический опыт в индустрии программного обеспечения, но и поможет снять ореол мистики с процесса моделирования программных систем. Мы полагаем, что наличие стандартного языка моделирования побудит большее число разработчиков моделировать программные системы еще до начала их построения. Быстрое и широкое распространение языка UML демонстрирует все большее признание преимуществ моделирования в сообществе разработчиков.

Создание языка UML представляло собой итеративный и расширяющийся процесс, очень похожий на моделирование большой программной системы. Конечным результатом этой работы является некий стандарт, построенный на основе многих идей и при участии большого количества людей и компаний из объектно-ориентированного сообщества. Мы начали разработку языка UML, однако многие другие помогли довести ее до успешного завершения, и мы благодарны им за вклад в общее дело.

Создание и согласование стандартного языка моделирования само по себе является серьезной задачей. Обучение сообщества разработчиков языку UML и представление его таким способом, который одновременно был бы доступен и соответствовал контексту процесса разработки программных систем, также является серьезной проблемой. В этой обманчиво краткой книге, дополненной с целью отразить самые последние изменения в языке UML, Мартин Фаулер оказался как никто более близок к решению этих проблем.

Мартин не только в ясной и доступной манере описывает ключевые аспекты языка UML, но также четко показывает ту роль, которую язык UML играет в процессе разработки. При прочтении книги мы получили истинное удовольствие от тех замечательных примеров моделирования, которые являются результатом более чем 12-летнего опыта работы Мартина в области проектирования и моделирования.

Данная книга служит введением в язык UML для многих тысяч разработчиков, пробуждая у них интерес к дальнейшему изучению преимуществ моделирования на основе теперь уже стандартного языка моделирования.

Мы рекомендуем эту книгу всем разработчикам, желающим познакомиться с языком UML и оценить перспективы той ключевой роли, которую он играет в процессе разработки.

Гради Буч,
Айвар Джекобсон,
Джеймс Рамбо

От автора

Два года назад издательство Addison-Wesley предложило мне написать книгу о новых особенностях языка UML. Хотя за прошедшее время и наблюдался огромный интерес к языку UML, изучить его можно было только по стандартной документации. Мы обработали много записей, чтобы быстро получить краткий вводный курс по новым аспектам языка UML, который бы послужил некоторым практическим руководством, пока год спустя не появятся более подробные и официальные книги.

Мы не ожидали, что эта книга появится после более детальных изданий. Большинство людей полагают, что если имеется выбор между кратким обзором и подробным описанием, то всякий выберет подробный текст. Не оспаривая эту общую точку зрения, я все же думаю, что даже при наличии подробных книг всегда останется место для сжатого обзора.

Два года спустя мои замыслы были реализованы в еще большей степени, чем я мог предположить. «UML в кратком изложении» стал бестселлером в области компьютерной индустрии. Хотя по языку UML появились хорошие подробные издания, эта книга все еще хорошо продается. Более того, изданы и другие обзорные книги, убеждая меня в том, что в мире с таким огромным количеством информации краткость все еще остается удачным выбором.

Итак, если все так прекрасно, зачем вам покупать эту книгу?

Я склонен считать, что вы уже слышали о языке UML. Этот язык стал стандартным способом изображения диаграмм в объектно-ориентированных проектах; он также распространяется и на не объектно-ориентированные области. Основные методы, существовавшие до языка UML, более не применяются. Язык UML добился признания и продолжает таким оставаться.

Если вы хотите изучить язык UML, эта книга – один из способов достичь этого. Главное преимущество этой книги в ее *небольшом* объеме. Купив большую книгу, вы получите больше информации, но вы будете также и дольше ее читать. Я выбрал наиболее важные части языка UML, так что вы ничего не потеряете. В этой книге вы найдете описание ключевых элементов нотации и их назначение. При желании двигаться дальше можно обратиться к более подробным изданиям.

Если вам понадобится более объемное руководство по языку UML, я рекомендую книгу Г. Буча (Grady Booch), Д. Рамбо (James Rumbaugh) и А. Джекобсона (Ivar Jacobson) «Язык UML. Руководство пользователя», 1999 [6], которая охватывает большую часть основ языка UML. В этой хорошо написанной книге объясняется применение языка UML для решения различных задач моделирования.

Как настоящая книга, так и «Руководство пользователя» предполагают, что вы уже кое-что знаете об объектно-ориентированной разработке. Хотя многие читатели говорили мне, что эта книга является хорошим введением в объекты, сам я так не считаю. Если вы хотите познакомиться с введением в объекты на языке UML, то лучше обратиться к книге К. Лармана (Craig Larman), 1998 [28].

Хотя главной темой моей книги является язык UML, я включил в нее материал, который существенно дополняет последний. Язык UML является только малой частью того, что необходимо знать для успешной работы с объектами. Я думаю, очень важно обратить внимание и на некоторые другие проблемы.

Наиболее важная из них связана с процессом разработки программного обеспечения. Язык UML разрабатывался с таким расчетом, чтобы быть независимым от этого процесса. Вы можете делать все, что только пожелаете; язык UML позволит наполнить содержанием все ваши диаграммы. Однако содержание диаграмм останется не слишком богатым без знания процесса, который обеспечивает им контекст. По моему мнению, именно процесс является важным фактором, и хороший процесс не должен быть сложным.

Поэтому я описал основные особенности процесса объектно-ориентированной разработки программного обеспечения. Это обеспечит нужный контекст для методов и поможет вам приступить к работе с объектами.

Другие темы включают в себя образцы, реорганизацию, самотестируемое программное обеспечение, проектирование по контракту и CRC-карточки. Ни одна из них не является частью языка UML, несмотря на это все они являются чрезвычайно важными методами, которые я постоянно использую.

Структура книги

В главе 1 описывается, что представляет собой язык UML, история его создания и причины, которые могут побудить вас к его применению.

В главе 2 обсуждается процесс объектно-ориентированной разработки. Хотя язык UML существует независимо от этого процесса, я думаю, что трудно обсуждать методы моделирования без упоминания о том, как они используются в процессе разработки.

В главах с 3 по 6 рассматриваются три наиболее важные метода языка UML: диаграммы вариантов использования, диаграммы классов и модели взаимодействия. Язык UML достаточно велик по объему, но вам нет необходимости знать его весь. Эти три метода являются ядром, которое необходимо практически каждому. Начните с них и дополните их другими методами по мере необходимости. (Следует отметить, что поскольку диаграммы классов сами по себе довольно сложны, ключевые понятия диаграммы классов рассматриваются в главе 4, а более сложные понятия – в главе 6.)

В главах с 7 по 10 излагаются остальные методы. Все они важны, но не для всякого проекта необходим каждый из этих методов. Таким образом, эти главы содержат достаточно информации, чтобы показать вам необходимость использования того или иного метода.

Для каждого метода описывается соответствующая нотация, объясняется, что означает эта нотация, и даются советы по использованию этих методов. Я стремлюсь сделать ясной спецификацию языка UML и в то же время донести до вас свое понимание того, как его использовать наилучшим образом. При этом даются ссылки на другие издания, которые позволят получить дополнительную информацию.

Глава 11 содержит описание небольшого примера, показывающего связь языка UML с программированием на языке (конечно) Java.

Изложение охватывает обзор нотации языка UML. Старайтесь применять ее по мере чтения глав, тем самым вы сможете проверить нотацию для различных понятий моделирования.

Если вы сочтете эту книгу интересной, то дополнительную информацию о моей работе по использованию языка UML, образцов и моделей можно найти на моей домашней странице в Интернете по адресу *http://ourworld.compuserve.com/homepages/Martin_Fowler*.

Изменения во втором издании

Поскольку язык UML развивался, я непрерывно обновлял содержание книги, учитывая при этом отзывы читателей о первом издании. Мы перепечатывали материал каждые два или три месяца; едва ли не каждый вариант рукописи содержал обновления, которые привели в результате к серьезному переутомлению в процессе нашей издательской деятельности.

Когда версия языка UML изменилась с 1.2 на 1.3, мы решили подвергнуть тщательной переработке всю книгу, что оказалось более чем достаточным для выпуска второго издания. Поскольку книга пользовалась популярностью, я старался не изменять ее характерные особенности. Я был весьма осторожен с добавлением нового материала. Я отбирал только важные с моей точки зрения вопросы.

Наибольшие изменения коснулись главы 3, посвященной вариантам использования, и главы 9, посвященной диаграммам деятельности. Эти главы были подвергнуты серьезной переработке. Также был добавлен подраздел по кооперации в главу 7. Я воспользовался прекрасной возможностью внести много мелких изменений и в другие части книги, основываясь на отзывах читателей и результатах моей работы за последние два года.

Благодарности первого издания

Столь быстрое издание книги потребовало большой помощи со стороны людей, которые готовили ее выпуск и приложили немало усилий, чтобы сделать все как можно скорее.

Кендалл Скотт сыграл важную роль в объединении всего материала, а также в работе над текстом и графикой. Когда я изменял что-либо в книге, он продолжал хранить все черновые наброски, прилагая при этом неимоверные усилия, чтобы справиться как с мелкими замечаниями, так и с невозможностью уложиться в заданные сроки.

«Трое друзей», Гради Буч, Айвар Джекобсон и Джим Рамбо, оказывали нам всевозможную поддержку и помощь советами. Мы потратили много часов на межконтинентальные телефонные разговоры, что позволило существенно улучшить нашу книгу (так же как и мое понимание языка UML).

Улучшению качества книги способствовала хорошая критика со стороны рецензентов. Последние не только предоставили в мое распоряжение необходимые отзывы, но и возвращали мне свои комментарии менее чем за неделю, чтобы не нарушить наши сжатые сроки. Приношу свою благодарность Симми Коччару Баргаве (Netscape Communications Corporation), Эрику Эвансу, Тому Хэдфилду (Evolve Software, Inc.), Рональду Джеффрису, Джошуа Кериевски (Industrial Logic, Inc.), Хелен Клейн (Университет Мичигана), Джеймсу Оделлу и Вивек Салгар (Netscape Communications Corporation). Двойное спасибо Тому Хэдфилду, поскольку он сделал это дважды!

Я хотел бы поблагодарить Джима Оделла за две вещи: во-первых, за координацию усилий Object Management Group (OMG) по созданию единого стандарта языка UML, который будет большим шагом вперед в нашем деле, и, во-вторых, за поддержку моей работы в области объектно-ориентированного анализа и проектирования. Я также благодарен ему за рецензирование этой книги.

Благодарю Синди за терпение, проявленное в мое отсутствие, даже когда я находился дома.

Не могу даже представить себе все трудности, с которыми столкнулись мой редактор, Дж. Картер Шэнклин, и его ассистент, Анжела Бьюнинг, чтобы выпустить книгу так быстро, как они это сделали. Каки-

ми бы ни были эти трудности, я уверен, что Картер и Анжела заслуживают моей особой благодарности. При издании книги вовсе не обязательно каждые два месяца перепечатывать рабочий вариант книги, но Картер и его команда проделали отличную работу, не придавая значения этому факту.

При подготовке материала для книги у меня часто возникали вопросы относительно отдельных особенностей языка UML. Конрад Бок, Айвар Джекобсон, Крис Кобрин, Джим Оделл, Гус Ремакерс и Джим Рамбо сделали все возможное, чтобы я смог найти ответы на все свои вопросы.

Большое количество читателей прислали мне сообщения с указанием различных ошибок и опечаток; их слишком много, чтобы перечислить каждого из них, но я благодарен им всем.

Наконец, но отнюдь не в последнюю очередь, я благодарен моим родителям за помощь в получении хорошего образования, с которого все и началось.

Мартин Фаулер
Мелроуз, Массачусетс,
Апрель 1999
fowler@acm.org,
http://ourworld.compuserve.com/homepages/Martin_Fowler

1

Введение

Что такое UML?

Унифицированный язык моделирования (UML, Unified Modeling Language) является преемником методов объектно-ориентированного анализа и проектирования (OOA&D), которые появились в конце 80-х и начале 90-х годов. Он непосредственно унифицирует методы Буча, Рамбо (OMT) и Джекобсона, однако обладает большими возможностями. Язык UML прошел процесс стандартизации в рамках консорциума OMG (Object Management Group) и в настоящее время является стандартом OMG.

UML – это название языка моделирования, но не метода. Большинство методов включают в себя, по крайней мере, в принципе, язык моделирования и процесс. **Язык моделирования** – это нотация (главным образом, графическая), которую используют методы для описания проектов. **Процесс** – это рекомендация относительно этапов, которые необходимо выполнить при разработке проекта.

Разделы многих книг по методам, в которых описывается процесс, выглядят очень поверхностно. Более того, по моему мнению, большинство людей, которые утверждают, что они используют метод, на самом деле используют язык моделирования, изредка соблюдая при этом требования процесса. Таким образом, зачастую язык моделирования является наиболее важной частью метода. Кроме того, язык, несомненно, является главным средством общения. Если вы хотите обсудить ваш проект с кем-либо еще, то каждый из вас должен понимать

именно язык моделирования, а *не* процесс, который использовался при разработке этого проекта.

«Трое друзей» также разработали некий унифицированный процесс, который они назвали Рациональный унифицированный процесс (RUP, Rational Unified Process). Для применения языка UML вовсе не обязательно использовать процесс RUP, поскольку они совершенно независимы. Тем не менее, в этой книге я описываю этот процесс с целью рассмотрения методов языка моделирования в некотором контексте. В рамках этого обсуждения используются основные этапы и терминология RUP, однако полное описание процесса RUP в книге не приводится. Должен сказать, что в своей работе мне приходится использовать много различных процессов, что зависит от заказчика и типа разрабатываемого программного обеспечения. Несмотря на то что я нахожу весьма важным стандартный язык моделирования, я не вижу такой же насущной необходимости в стандартном процессе, хотя некоторое согласование терминологии все же будет полезным.

Как мы к этому пришли

В 80-х годах объекты начали выходить из исследовательских лабораторий и делать свои первые шаги в направлении «реального» мира. Язык Smalltalk был реализован на некоторой платформе и стал пригодным для практического использования; появился на свет и C++.

Подобно многим другим разработкам в области программного обеспечения, популярности объектов способствовали языки программирования. Многие люди были удивлены той быстротой, с какой методы проектирования смогли проникнуть в объектно-ориентированный мир. Методы проектирования приобрели большую популярность в промышленных разработках 70-х и 80-х годов. Многие считали, что методы, помогающие эффективно осуществлять анализ и проектирование, в такой же степени важны и для объектно-ориентированной разработки.

Между 1988 и 1992 годами появились следующие основные книги, посвященные методам объектно-ориентированного анализа и проектирования:

- Салли Шлеер (Sally Shlaer) и Стив Меллор (Steve Mellor) написали пару книг (1989 [40] и 1991 [41]) на тему анализа и проектирования; материал этих книг со временем воплотился в их метод рекурсивного проектирования (Recursive Design), 1997 [42].
- Питер Коуд (Peter Coad) и Эд Йордон (Ed Yourdon) также написали книги, в которых был разработан неформальный и ориентированный на прототипирование метод Коуда. См. Коуд и Йордон, 1991 [9] и [10], Коуд и Никола, 1993 [8], Коуд и др., 1995 [11].
- Разработчики языка Smalltalk в Портленде (Орегон) выступили с двумя методами: метод проектирования на основе ответственностей

(Responsibility-Driven Design) Вирс-Брок (Wirfs-Brock) и др., 1990 [46] и CRC-карточки (Class-Responsibility-Collaboration) Бек (Beck) и Каннингхем (Cunningham), 1989 [3].

- Гради Буч из компании Rational Software выполнил большую работу, связанную с разработкой систем на языке Ada. Его книги содержат несколько примеров (и лучшие карикатуры о методах). См. книги Буча, 1994 [4] и 1996 [5].
- Джим Рамбо возглавил группу в исследовательской лаборатории General Electric и написал очень популярную книгу о методе, который получил название Object Modeling Technique (OMT). См. книгу Рамбо и др., 1991 [38], а также Рамбо, 1996 [36].
- Джим Оделл (Jim Odell) совместно с Джеймсом Мартином (James Martin) написал свои книги на основе достаточно большого опыта создания информационных систем в области бизнеса и использования информационных технологий. Из всех перечисленных книг результаты его работы носят наиболее концептуальный характер. См. книгу Мартина и Оделла, 1998 [29].
- Айвар Джекобсон построил материал своих книг на основе своего опыта работы с телефонными системами фирмы Ericsson и впервые ввел понятие варианта использования (use case). См. книги Джекобсона, 1992 [24] и 1995 [25].

В период моей подготовки к поездке в Портленд на конференцию OOPSLA'94 среди методов анализа и проектирования наблюдалось сильное разобщение и конкуренция. В то время каждый из вышеупомянутых авторов являлся неформальным лидером группы разработчиков-практиков, которые разделяли его идеи. Все эти методы были очень похожи друг на друга, тем не менее, между ними существовали досадные различия во второстепенных деталях. Графические нотации одних и тех же основных понятий могли существенно различаться, что вызывало путаницу у моих заказчиков.

Очевидно, назревал разговор на тему стандартизации, но никто, казалось, не собирался ничего предпринимать для этого. Некоторые даже выступали против самой идеи стандартизации методов. Другим эта идея нравилась, но они не были готовы прилагать какие-либо усилия в этом направлении. Команда из OMG попыталась взяться за решение проблемы стандартизации, но в ответ получила только открытое письмо с протестом от всех авторов основных методологий. Попытка Гради Буча предложить неформальное обсуждение за чашкой утреннего кофе не увенчалась особым успехом. (Это напоминает мне одну старую шутку. Вопрос: какая разница между автором методологии и террористом? Ответ: с террористом можно договориться.)

Для сообщества специалистов по объектно-ориентированным методам большой новостью на конференции OOPSLA'94 стал тот факт, что Джим Рамбо покинул фирму General Electric и присоединился к Гради

Бучу из компании Rational Software с намерением объединить их методы.

Следующий год прошел в невольном удивлении.

Гради и Джим провозгласили, что «война методов закончилась – мы победили», объявив по существу, что они собираются достичь стандартизации способом фирмы Microsoft. Ряд других методологов предложил сформировать анти-Бучевскую коалицию.

К конференции OOPSLA'95 Гради и Джим подготовили первое общедоступное описание своего объединенного метода в форме документации на *Унифицированный метод* (*Unified Method*) версии 0.8. Однако более важным оказалось их сообщение о приобретении фирмой Rational Software компании Objectory и о том, что Айвар Джекобсон присоединится к команде разработчиков Унифицированного метода. Фирма Rational устроила презентацию подготовленной версии 0.8, которая была воспринята с большим вниманием. Встреча прошла достаточно весело, ее не смогло испортить даже пение Джима Рамбо.

Гради, Джим и Айвар, получившие широкую известность как «трое друзей», в течение 1996 года продолжали работу над своим методом, но уже под новым названием: Унифицированный язык моделирования (UML). Однако другие главные действующие лица из сообщества разработчиков объектных методов не были расположены к тому, чтобы последнее слово осталось за UML.

Для стандартизации этих методов в рамках консорциума OMG была сформирована инициативная группа. Эта попытка решить данную проблему оказалась гораздо более серьезной, чем предыдущие усилия OMG в области этих методов. Председателем группы была выбрана Мэри Лумис (Mary Loomis), а затем в работу в качестве сопредседателя включился Джим Оделл, став фактическим лидером этой деятельности. Оделл дал ясно понять, что не желает поддерживать стандарт, предлагаемый компанией Rational, и готов предложить в качестве стандарта свой собственный метод.

В январе 1997 года различные организации представили на рассмотрение свои предложения по стандартизации методов, что способствовало обмену информацией между различными моделями. Эти предложения касались в основном метамодели и необязательной нотации. Среди этих предложений была и документация на язык UML версии 1.0, подготовленная компанией Rational.

После этого последовал короткий, но трудный период времени, в течение которого были объединены различные предложения. В результате консорциум OMG выдвинул в качестве официального стандарта OMG версию 1.1 языка UML. В последующем инициативная группа по пересмотру (RTF) языка UML, возглавляемая Крисом Кобрином (Cris Kobryn), внесла в язык несколько существенных дополнений. В то время как версия 1.2 мало отличалась от своей предшественницы, версия 1.3,

ставшая общедоступной в начале 1999 года, подверглась серьезным изменениям. Весной 1999 года эксперты из инициативной группы RTF завершили свою работу, а версия 1.3 стала очередной официальной версией языка UML.

Нотации и метамодели

Язык UML, в своем нынешнем состоянии, определяет нотацию и метамодель.

Нотация представляет собой совокупность графических элементов, которые используются в моделях; она является синтаксисом данного языка моделирования. Например, нотация диаграммы классов определяет, каким образом представляются такие элементы и понятия, как класс, ассоциация и кратность.

Конечно, при этом возникает вопрос точного определения смысла ассоциации, кратности и даже класса. Обычное употребление этих понятий предполагает некоторые неформальные определения, однако многие разработчики испытывают потребность в более строгом определении соответствующих понятий.

Идея строгих языков для спецификации и проектирования является наиболее распространенной в области формальных методов. В таких методах проекты и спецификации представляются с использованием некоторых средств исчисления предикатов. Соответствующие определения являются математически строгими и исключают неоднозначность. Однако эти определения нельзя считать универсальными. Даже если вы сможете доказать, что программа соответствует математической спецификации, не существует способа доказать, что эта математическая спецификация действительно удовлетворяет реальным требованиям системы.

Проектирование должно основываться на всестороннем анализе всех ключевых вопросов разработки. Использование формальных методов зачастую приводит к тому, что проектирование тонет во множестве мелких второстепенных деталей. Кроме того, формальные методы сложны для понимания и применения, и работать с ними зачастую труднее, чем с языками программирования. К тому же формальные методы не могут исполняться, как программы.

Большинство объектно-ориентированных методов являются отнюдь не строгими; их нотация в большей степени апеллирует к интуиции, чем к формальному определению. В целом, это не выглядит таким уж большим недостатком. Хотя подобные методы могут быть неформальными, многие разработчики по-прежнему считают их полезными, и это нельзя не принимать во внимание.

Разработчики объектно-ориентированных методов ищут способы добиться большей строгости методов, не жертвуя при этом их практичес-

кой полезностью. Один из способов заключается в определении некоторой **метамодели**: диаграммы (как правило, диаграммы классов), определяющей нотацию.

На рис. 1.1 изображена небольшая часть метамодели языка UML, которая показывает отношение между ассоциациями и обобщением. (Этот фрагмент приведен с единственной целью – дать лишь общее представление о том, что такое метамодель. Я даже не буду пытаться давать здесь какие-либо дополнительные пояснения.)

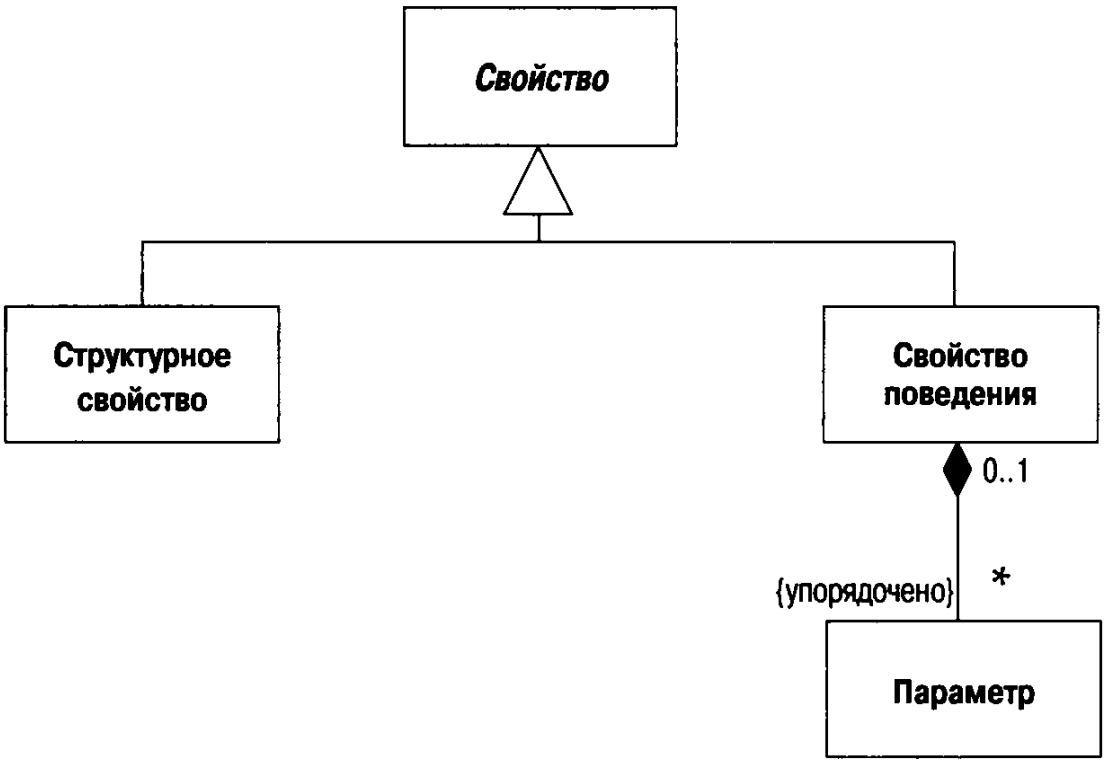


Рис. 1.1. Фрагмент метамодели языка UML

Насколько велико влияние метамодели на того, кто использует соответствующую нотацию при моделировании? Как минимум, она помогает определить, что такое правильно построенная модель, то есть модель, являющаяся синтаксически правильной. Кроме того, пользователь, обладающий высокой квалификацией в области методов, должен понимать данную метамодель. Однако большинству пользователей методов вовсе не нужно столь глубокое понимание метамодели для практического применения нотации языка UML.

Именно поэтому изложение материала в книге не является формальным; я следую по пути традиционных методов и апеллирую к вашей интуиции. При желании получить более строгое изложение обратитесь к справочному руководству или официальной документации.

Насколько глубоко вам придется погрузиться в изучение данного языка моделирования? Это зависит от цели, с которой вы его используете. Если вы располагаете CASE-средством, которое генерирует код, то для получения эффективного кода необходимо хорошо понимать механизм интерпретации языка моделирования этим CASE-средством. Ес-

ли же вы используете диаграммы только для общения или обмена информацией между разработчиками, то может оказаться достаточным гораздо меньший объем знаний.

Если не придерживаться официальной нотации, другие разработчики и вовсе могут не понять ваши модели. Однако в отдельных случаях официальная нотация может оказаться не в состоянии удовлетворить все ваши потребности. Должен признаться, что в подобных ситуациях я не боюсь вносить в язык некоторые дополнения. Полагаю, что дополнение или уточнение языка вполне оправдано, если это помогает улучшить обмен информацией между разработчиками. Но я не часто прибегаю к подобным действиям, поскольку всегда осознаю, что дополнение и уточнение языка может служить источником проблем. В этой книге специально оговариваются те места, где приводятся небольшие дополнения или уточнения языка UML.

Для чего нужно заниматься анализом и проектированием

В процессе работы следует всегда помнить, что самым главным в разработке программного обеспечения является написание кода. При всем этом диаграммы представляют собой всего лишь красивые картинки. Ни один пользователь не станет благодарить вас за красивые картинки; ему нужны работающие программы.

Поэтому, если вы собираетесь использовать язык UML, очень важно задать себе вопрос, зачем это нужно и как это поможет при написании программного кода. Хотя нет подходящего эмпирического доказательства того, насколько хороши это методы или плохи, однако в следующих подразделах рассматриваются причины, по которым я часто прибегаю к использованию этих методов.

В этом разделе мы лишь кратко остановимся на методах, которые более подробно будут описаны далее в книге. Если вам покажется что-либо непонятным, пропустите этот раздел и вернитесь к нему позже.

Общение

Основной причиной использования языка UML является общение разработчиков между собой. Язык UML позволяет сообщать о своих идеях более наглядно, нежели другие способы. Естественный язык слишком неточен и приводит к путанице, когда он сталкивается с более сложными понятиями. Программный код точен, но слишком насыщен деталями реализации. Таким образом, я использую язык UML, когда мне необходима вполне определенная точность, но не нужны излишние детали. Но это вовсе не означает, что при этом не рассматрива-

ются детали. Скорее наоборот, я использую язык UML с целью выдвижения на первый план именно *важных* деталей.

В качестве консультанта мне зачастую приходится вникать в сложные проекты и какое-то время выглядеть интеллектуалом. Язык UML становится бесценным средством в этой работе, поскольку помогает получить некоторое представление о системе в целом. Только взглянув на диаграмму классов, можно быстро получить представление о том, какие виды абстракций существуют в этой системе и где расположены наименее проработанные части модели, требующие последующего уточнения. При дальнейшем знакомстве с системой необходимо знать, как классы кооперируются между собой. Для этого рассматриваются диаграммы взаимодействия, которые иллюстрируют основные аспекты поведения системы.

Если это оказывается полезным для меня как внешнего консультанта, то это также полезно для команды разработчиков проекта. Очень легко потерять обзор в лесу за деревьями большого проекта. Выбрав для себя несколько диаграмм, вы легче отыщете свой путь в дебрях программного обеспечения.

Для построения карты дорог большой системы можно использовать диаграмму пакетов (см. главу 7), которая изображает главные составные части системы и зависимости между ними. После чего для каждого пакета можно нарисовать диаграмму классов. Когда будете рисовать диаграмму классов в этом контексте, используйте точку зрения спецификации. Очень важно на этом этапе работы скрыть детали реализации. При этом следует также нарисовать диаграммы взаимодействия для основных взаимодействий в отдельном пакете.

Если некоторые элементы системы многократно встречаются в вашей модели, то для важнейших из них следует использовать образцы (см. врезку в главе 2). С их помощью можно объяснить, почему ваш проект выполнен так, а не иначе. Также полезно описать проекты, которые вы отклонили, и почему это было сделано. Заканчивая работу, я постоянно забываю о решениях подобного рода.

Следуя этим рекомендациям, кратко фиксируйте все результаты. Важнейшая часть коммуникации состоит в выделении главных особенностей вашей работы. Нет никакой необходимости показывать каждое свойство каждого класса; вместо этого следует указать лишь наиболее важные детали. С точки зрения обмена информацией краткий документ намного лучше большого по объему, а понимание того, что следует исключить из описания, является искусством.

Изучение объектно-ориентированных методов

Многие разработчики отмечают наличие серьезных трудностей, связанных с изучением объектно-ориентированного подхода, и, в первую очередь, с пользующейся плохой славой сменой парадигмы. В некото-

рых случаях переход к объектно-ориентированным методам происходит легко. В других случаях при работе с объектами приходится сталкиваться с рядом препятствий, особенно при желании использовать все их потенциальные возможности.

Это вовсе не означает, что так уж трудно научиться разрабатывать программы на каком-либо объектно-ориентированном языке программирования. Проблема заключается в том, чтобы научиться использовать преимущества объектно-ориентированных языков. Том Хэдфилд (Tom Hadfield) удачно сформулировал это следующей фразой: объектные языки *обладают* преимуществами, но не *предоставляют* их автоматически. Чтобы использовать эти преимущества, вы должны пойти на смену пользующейся плохой славой парадигмы. (Немедленно убедитесь, что вы все еще сидите!)

Средства языка UML разрабатывались в некоторой степени для того, чтобы помочь людям выполнять качественные объектно-ориентированные проекты, однако разные средства обладают различными достоинствами.

- Одним из наилучших способов изучения объектно-ориентированных методов являются CRC-карточки (см. врезку в главе 5). Они не являются составной частью языка UML, хотя могут и должны использоваться вместе с ним. CRC-карточки были созданы главным образом для обучения людей работе с объектами. Именно поэтому они намеренно отличаются от традиционных методов проектирования. Запись на этих карточках ответственности класса и отсутствие сложной нотации делает их особенно важным средством.

- Диаграммы взаимодействия (см. главу 5) оказываются очень полезными, поскольку позволяют наглядно представить структуру сообщения и тем самым выявить излишне централизованные проекты, в которых один объект выполняет всю работу.

- Диаграммы классов (см. главы 4 и 6), используемые для иллюстрации моделей классов, одновременно как хороши, так и плохи для изучения объектов. Модели классов столь же удобны, как и модели данных; многие принципы построения хорошей модели данных также пригодны для построения хорошей модели классов. Основная проблема в использовании диаграмм классов состоит в том, что можно легко разработать модель классов, которая ориентирована на данные, а не на ответственность.

- Концепция образцов (см. врезку в главе 2) стала жизненно необходимой при изучении объектно-ориентированных методов, поскольку предоставляет возможность использовать результаты хорошо выполненных объектно-ориентированных проектов и обучаться на их примерах. После того как вы освоили некоторые базовые методы моделирования, такие как простые диаграммы классов и диаграм-

мы взаимодействия, самое время приступить к рассмотрению образцов.

- Еще одним важным методом является итеративная разработка (см. главу 2). Этот метод не поможет вам в изучении объектно-ориентированного подхода непосредственно, однако он даст ключ к его эффективному использованию. Если вы применяете итеративную разработку с самого начала, то сможете правильно уяснить для себя характер процесса и начнете понимать, почему проектировщики предлагают делать некоторые вещи именно так, а не иначе.

Применяя определенный метод, старайтесь пользоваться данной книгой. Я рекомендую начинать с простых нотаций, в частности с диаграмм классов. Затем можно перейти к изучению более сложных понятий, которые представляются вам необходимыми. Возможно, вы придете к выводу, что соответствующий метод следует расширить.

Общение с экспертами предметной области

Одно из важнейших требований при разработке заключается в построении *правильной* системы, то есть такой системы, которая удовлетворяет потребности пользователей за разумную стоимость. Добиться этого весьма трудно, поскольку мы, с нашим жаргоном, должны общаться с пользователями, которые имеют собственный, еще более непонятный жаргон. (Я довольно продолжительное время работал в области медицины, где жаргон имеет весьма мало общего с английским языком!) Установление хороших контактов с пользователями наряду с хорошим пониманием их мира является ключом к разработке качественного программного обеспечения.

Очевидным методом, который следует применять для этого, являются варианты использования (см. главу 3). **Вариант использования** представляет собой некий моментальный снимок одного из аспектов вашей системы. Совокупность всех вариантов использования образует внешнее представление вашей системы, которому предстоит пройти долгий путь, объясняя, что эта система будет делать.

Хороший набор вариантов применения – это главное для понимания потребностей ваших пользователей. Варианты использования также представляют собой хорошее средство для планирования проекта, поскольку они позволяют управлять итеративной разработкой. Последняя сама является важным методом, так как обеспечивает регулярную обратную связь с пользователями и тем самым позволяет отслеживать текущее состояние разработки программного обеспечения.

Хотя варианты использования и помогают установить взаимопонимание в отношении внешнего представления системы, исключительно важно рассмотреть также и более глубокие аспекты. Имеется в виду изучение того, как эксперты в вашей предметной области понимают свой мир.

В данной ситуации весьма важными могут оказаться диаграммы классов (см. главы 4 и 6), пока вы рассматриваете их с *концептуальной* точки зрения. Другими словами, каждый класс следует трактовать как некоторое понятие из области мышления пользователя. Построенные в этом случае диаграммы классов не являются диаграммами данных или классов, а представляют собой, скорее, диаграммы языка ваших пользователей.

В тех случаях, когда важной частью предметной области пользователей являются потоки работ, я считаю весьма полезным использовать диаграммы деятельности (см. главу 9). Поскольку диаграммы деятельностей поддерживают параллельные процессы, они позволяют избавиться от тех из них, выполнение которых не является необходимо последовательным. Характерной особенностью этих диаграмм является преуменьшение роли их связей с классами, что может повлечь за собой проблемы на более поздних стадиях проектирования, но на концептуальном этапе процесса разработки создает определенные преимущества.

Где найти дополнительную информацию

Эта книга вовсе не является полным и окончательным справочным руководством по языку UML, не говоря уже об объектно-ориентированном анализе и проектировании. Существует много такого, о чем здесь не говорится, и много полезного, что следует дополнительно прочесть. В ходе рассмотрения отдельных тем будут упоминаться и другие книги, к которым следует обратиться для получения более подробной информации о языке UML и объектно-ориентированном анализе, а также о проектировании в целом.

Без сомнения, после прочтения данного справочного руководства необходимо обратиться к книгам по языку UML, написанным «тремя друзьями»:

- Гради Буч возглавил работу по написанию руководства пользователя (Буч, Рамбо и Джекобсон, 1999 [6]). Эта книга является учебником, содержащим описание практических аспектов использования языка UML для решения различных задач проектирования.
- Джим Рамбо возглавил деятельность по написанию справочного руководства (Рамбо, Джекобсон и Буч, 1999 [37]). Я часто обращаюсь за помощью к этому детальному справочнику по языку UML.
- Айвар Джекобсон возглавил работу над книгой по описанию процесса, в ходе которого используется язык UML (Джекобсон, Буч и Рамбо, 1999 [23]). Более подробно об этом процессе говорится в главе 2.

Конечно, чтобы хорошо изучить объектно-ориентированный анализ и проектирование, далеко не достаточно прочесть только эти книги

«троих друзей». Мой список рекомендуемой литературы часто изменяется; самую последнюю информацию можно получить на моей домашней странице в Интернете.

Если вы новичок в объектном подходе, то обратитесь к моему фавориту среди имеющихся справочников и руководств для начинающих – книге Ларман, 1998 [28]. Ее достоинством является то, что автор следует строго управляемому ответственностью подходу к проектированию. Для тех, кто хочет больше знать об объектах с концептуальной точки зрения, теперь в серии книг по языку UML есть работа Мартина и Оделла, 1998 [29]. Разработчикам-практикам следует обратиться также к книге Дугласа (Douglass), 1998 [17].

Для расширения вашего кругозора также рекомендую прочитать книги по образцам. Сейчас, когда война методов уже закончилась, я думаю, что именно образцы окажутся среди наиболее интересных материалов по анализу и проектированию. Поскольку разработчики непременно будут создавать новые методы анализа и проектирования, то, вероятно, со временем они объяснят, как эти методы могут использоваться вместе с языком UML. В этом еще одно преимущество языка UML; он поощряет разработчиков создавать новые методы без дублирования уже выполненной кем-то работы.

2

Основы процесса разработки

UML – это язык моделирования, а вовсе не метод. Язык UML не содержит понятие процесса, который является важной частью метода.

Название этой книги – «UML. Основы», поэтому можно было бы совершенно спокойно игнорировать сам процесс. Тем не менее, я думаю, что методы моделирования не имеют смысла без знания того, как они могут быть использованы в процессе. Я решил рассмотреть процесс в первую очередь, чтобы вы могли представить, каким образом выполняется объектно-ориентированная разработка. Но это не попытка глубоко проникнуть в многочисленные детали этого процесса, а лишь описание его основных особенностей. Знания последних вполне достаточно для типичного использования этих методов при разработке проекта.

«Трое друзей» разработали некий единый процесс, получивший название *Rational Unified Process*. (Ранее я использовал в его названии слово *Objectory*.) Этот процесс описан в книге «трех друзей» (Джекобсон, Буч и Рамбо, 1999 [23]).

Рассматривая основы процесса, я буду пользоваться терминологией и базовыми понятиями Рационального унифицированного процесса (Rational Unified Process). Однако я не пытаюсь дать описание самого RUP, поскольку это выходит за рамки данной книги. Взамен приведу лишь достаточно поверхностное и неформальное описание процесса, которое согласуется с RUP. Тем, кто заинтересуется всеми деталями Рационального унифицированного процесса, следует обратиться к

книге «троих друзей», посвященной процессу [23], или к обзору Крухтена (Kruchten), 1999 [27].

Хотя Рациональный унифицированный процесс содержит детальное описание того, какого рода модели следует разрабатывать на различных фазах процесса, я не буду углубляться в такие детали. Также не будут описываться задачи, ресурсы и роли. Моя терминология беднее, чем терминология RUP; это та цена, которую приходится платить за поверхностное описание.

Какой бы процесс ни рассматривался, не забывайте, что с языком UML можно использовать *любой* процесс. Язык UML не зависит от процесса. Следует выбрать тот, который подходит для вашего типа проекта. Какой бы процесс вы ни применяли, язык UML может использоваться для записи полученных решений по анализу и проектированию.

В действительности я вовсе не думаю, что можно пользоваться только одним процессом для разработки программного обеспечения. Различные факторы, связанные с разработкой программного обеспечения, приводят к различным типам процессов. Эти факторы относятся к типу разрабатываемого программного обеспечения (программа, работающая в реальном времени, информационная система или настольный продукт), масштабности разработки (один разработчик, небольшая команда, корпоративная разработка) и т. д.

По моему мнению, командам следует применять свои собственные процессы, используя известные процессы не в качестве стандартов, а только лишь как рекомендации.

Общее представление о процессе

На рис. 2.1 изображено самое общее представление о процессе разработки.

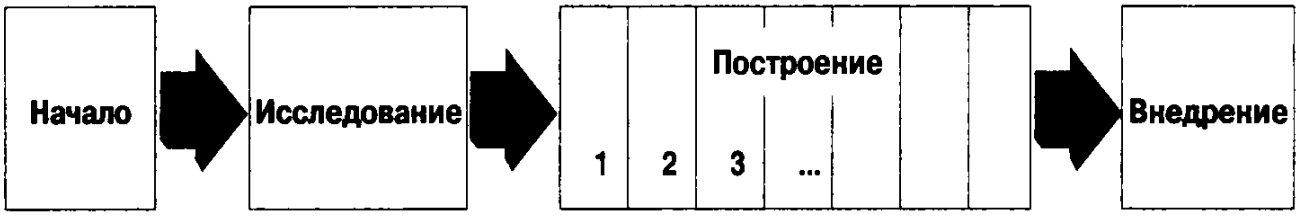


Рис. 2.1. Схема процесса разработки

Это итеративный и нарастающий процесс, при котором программное обеспечение не создается одним большим ударом в конце проекта, а, напротив, разрабатывается и реализуется по частям. Фаза **построения** состоит из многих **итераций**, на каждой из которых выполняются построение, тестирование и интеграция высококачественного программного обеспечения, удовлетворяющего некоторому подмножеству тре-

бований к проекту. Передача разработанного программного обеспечения в эксплуатацию может быть как внешней для первых пользователей, так и чисто внутренней. Каждая итерация содержит все обычные фазы жизненного цикла программного обеспечения: анализ, проектирование, реализация и тестирование.

В принципе вы можете начать с самого начала: выбрать некоторую функциональность и реализовать ее, затем выбрать другую функциональность и т. д. Однако полезно потратить некоторое время на планирование разработки.

Первыми двумя фазами являются начало и исследование. В **начальной** фазе разрабатывается экономическое обоснование проекта и определяются его границы. Именно на этой фазе спонсор проекта принимает на себя определенные обязательства относительно дальнейшей работы. В фазе **исследование** уточняются более детально требования, выполняются высокоуровневый анализ и проектирование для построения базовой архитектуры и создается план для фазы построения.

Даже для такого рода итеративного процесса имеется некоторая работа, которая должна быть выполнена в самом конце – в фазе **внедрения**. Эта работа может включать бета-тестирование, оптимизацию производительности и обучение пользователей.

Проекты отличаются между собой количеством необходимых **формальностей**. Сильно формализованные проекты требуют множества официальных отчетов, совещаний и согласований. В слабо формализованных проектах начальная фаза может состоять из часовой беседы со спонсором проекта и построения плана, который помещается на одной странице. Естественно, чем больше проект, тем больше он требует формальностей. Основные принципы всех фаз должны соблюдаться всегда, независимо от того, каким способом реализуется проект.

Лично я пытаюсь свести эти формальности к минимуму, и мое изложение отражает это стремление. Существует изобилие излишне формализованных процессов, которые можно найти где угодно.

Ранее итерации рассматривались лишь по отношению к одной фазе – фазе построения. На самом деле итерации могут осуществляться на всех фазах и часто бывают полезным средством для выполнения большой фазы. Однако именно построение является ключевой фазой, которая разбивается на итерации.

Таким образом, мы получили некоторое высокоуровневое представление о процессе разработки программного обеспечения. Теперь перейдем к деталям, чтобы получить достаточно информации о том, какое место занимают в общей картине методы, рассматриваемые далее в этой книге. По ходу изложения я немного расскажу о самих методах и о том, когда их использовать. У тех, кто не знаком с этими методами, могут возникнуть некоторые затруднения. В этом случае пропустите непонятное место и вернитесь к нему позже.

Начало

Начальная фаза может принимать множество различных форм. Для некоторых проектов это может быть беседа за чашкой кофе: «Давайте рассмотрим размещение каталога наших услуг в Интернете». Для более крупных проектов начальная фаза может превратиться во всестороннее изучение всех вариантов реализации проекта, которое займет месяцы.

На начальной фазе разрабатывается бизнес-план проекта – определяется, какова его приблизительная стоимость и размер ожидаемого дохода. Вам следует также определить границы проекта и выполнить некоторый предварительный анализ, чтобы представить себе его размеры.

Я вовсе не стремлюсь к тому, чтобы все выполнить в начальной фазе. Начальная фаза должна занимать несколько дней работы, в течение которых следует решить, стоит ли проводить в фазе исследования более глубокий анализ, который может затянуться на несколько месяцев (см. следующий подраздел). В этот момент спонсор проекта соглашается лишь на серьезное предварительное рассмотрение проекта.

Исследование

Итак, проект утвержден, и начинается его выполнение. Как правило, на этой фазе у вас имеется только нечеткое представление относительно требований к системе. Например, можно сказать следующее:

> *Мы собираемся создать систему следующего поколения для заказчиков компании Watts Galore Utility Company и намерены использовать объектно-ориентированную технологию для построения более гибкой системы, которая будет в большей степени ориентирована на пользователей, в частности такую, которая будет поддерживать их консолидированные счета.*

Вероятнее всего, ваше документальное описание требований будет более объемным, чем это, но в действительности оно не будет намного более содержательным.

В этот момент желательно попытаться лучше понять суть проблемы:

- Что вы на самом деле собираетесь создать?
- Как вы собираетесь это сделать?

Решая, какие вопросы рассматривать во время этой фазы, следует исходить, главным образом, из тех рисков, которые оказывают влияние на ваш проект. Что может привести к провалу проекта? Чем больше риск, тем большее внимание ему следует уделить.

Исходя из моего опыта, риски полезно классифицировать на четыре категории:

1. Риски, связанные с *требованиями*. Каковы требования к системе? Большая опасность заключается в том, что вы построите совсем не ту систему, которая будет выполнять вовсе не то, что нужно пользователям.
2. *Технологические риски*. С какими технологическими рисками вам придется столкнуться? Действительно ли позволяет выбранная вами технология реализовать ваш проект? Каким образом следует интегрировать различные части проекта?
3. Риски, связанные с *квалификацией персонала*. Сможете ли вы подобрать штат сотрудников с необходимым опытом и квалификацией?
4. *Политические риски*. Существуют ли политические силы, которые могут оказаться на вашем пути и серьезно повлиять на выполнение вашего проекта?

В вашем случае рисков может быть и больше. Но те риски, которые попадают в эти четыре категории, присутствуют почти всегда.

Риски, связанные с требованиями

Анализ требований к системе очень важен, и это именно та область, в которой методы языка UML позволяют получить наиболее очевидные результаты. Отправной точкой при этом являются варианты использования, которые управляют процессом разработки в целом.

Более детально варианты использования будут рассмотрены в главе 3, здесь же приводится лишь краткое описание их назначения.

Вариант использования отражает типичное взаимодействие пользователя с системой для достижения некоторой цели. Представьте себе текстовый процессор, который я использую в данный момент. Одним из вариантов использования может быть «проверка правописания», другим – «создание предметного указателя для документа».

Важной особенностью вариантов использования является то, что каждый из них определяет некоторую функцию, которая понятна пользователю и имеет для него определенное значение. Разработчик может отреагировать, например, следующим образом:

> *На реализацию функции создания предметного указателя у меня уйдет два месяца. Есть также вариант использования, связанный с поддержкой грамматической проверки. Думаю, это может занять три месяца. В нашем распоряжении только три месяца на выполнение проекта – что вы хотели бы получить в первую очередь?*

Варианты использования являются основой для общения и взаимопонимания между пользователями и разработчиками при планировании проекта.

Одна из наиболее важных вещей, которые необходимо сделать на фазе исследования, – это определение всех возможных вариантов использования разрабатываемой вами системы. На практике, разумеется, вы не в состоянии определить абсолютно все варианты. Однако следует выделить самые важные из них, которые в наибольшей степени подвержены рискам. Именно по этой причине на фазе исследования вы должны запланировать встречу с пользователем с целью формирования вариантов использования.

Варианты использования не следует излишне детализировать. Обычно я считаю, что вполне достаточно для этого текстового описания размером от одного до трех абзацев. Сам этот текст должен быть достаточно содержательным как для пользователей, чтобы они были в состоянии понять основную идею, так и для разработчиков, чтобы они хорошо представляли скрытый смысл этого описания.

Однако варианты использования не дают законченное представление о картине в целом. Другой важной задачей на этой фазе является разработка общей схемы концептуальной модели предметной области. При этом сама картина бизнес-деятельности находится в голове у одного или нескольких пользователей. Например:

> *Наши клиенты могут располагаться в нескольких различных пунктах, и в каждом из этих пунктов мы предоставляем набор услуг. В определенный момент времени клиент получает счет за все услуги, оказанные в данном пункте. Нам нужно, чтобы клиенту выставлялся счет за все услуги, оказанные во всех пунктах. Мы называем это консолидированным счетом.*

Этот фрагмент содержит слова «клиент», «пункт» и «услуга». Что означают эти термины? Как они соотносятся друг с другом? Концептуальная модель предметной области дает начало ответам на эти вопросы и в то же время закладывает основу модели объектов, которая будет позже использоваться в процессе разработки для представления объектов системы. Я использую термин **модель предметной области** для определения любой модели, главное назначение которой состоит в описании той части реального мира, в рамках которого создается компьютерная система. Такая модель не зависит от стадии, на которой находится процесс разработки.

Заметим, что Рациональный унифицированный процесс определяет термин «модель предметной области» более тщательно. Эта тема детально изложена в книге Джекобсона, Буча и Рамбо, 1999 [23]. Я придерживаюсь точки зрения большинства известных мне разработчиков из объектного сообщества.

Для построения модели предметной области особенно полезны следующие два метода языка UML.

Основным методом, который я использую для построения моделей предметной области, является диаграмма классов, рассматриваемая с концептуальной точки зрения (см. главу 4). Эти диаграммы могут быть использованы для представления понятий, которые применяют бизнес-аналитики при исследовании соответствующей бизнес-системы, и для представления особенностей объединения этих понятий в единой модели. Во многих случаях именно диаграммы классов определяют некоторый терминологический словарь для описания соответствующей предметной области.

Если предметная область также обладает явно выраженным элементом потока работ, я предпочитаю представлять этот аспект с помощью диаграмм деятельности (см. главу 9). Главной особенностью диаграмм деятельности является их способность изображать параллельные процессы, которые очень важны для исключения ненужных последовательностей в бизнес-процессах.

Некоторые разработчики предпочитают пользоваться диаграммами взаимодействия (см. главу 5) для исследования взаимодействия различных ролей в бизнес-системе. Рассматривая совместно исполнителей и деятельности, они приходят к лучшему пониманию соответствующего бизнес-процесса. Лично я предпочитаю использовать диаграммы деятельности для изображения того, что необходимо сделать в первую очередь, а также чтобы определить, кто и что будет делать впоследствии.

Моделирование предметной области может оказаться весьма важным дополнением вариантов использования. В процессе выявления и описания вариантов использования я предпочитаю предъявлять их эксперту предметной области и анализировать его реакцию на свое понимание соответствующей бизнес-системы. При этом целесообразно прибегать к помощи концептуальных диаграмм классов и диаграмм деятельности.

В такой ситуации можно обойтись минимумом обозначений, совсем не беспокоясь о строгости и делая на диаграмме множество поясняющих примечаний. Также нет необходимости фиксировать каждую деталь. Вместо этого нужно сосредоточиться на других более важных проблемах и областях предполагаемого риска. Это позволяет изображать множество несвязанных диаграмм, не беспокоясь при этом о согласованности и взаимозависимостях между ними.

После рассмотрения большинства существенных вопросов, необходимо объединить различные диаграммы в единую согласованную модель предметной области. С этой целью я использую одного или двух экспертов предметной области, у которых есть желание поглубже вникнуть в процесс моделирования. На этом этапе важно придерживаться

концептуальной точки зрения на проект, но в то же время придать ему большую строгость.

После этого построенная модель может быть использована в качестве начального представления для построения классов на фазе построения. Если модель получилась большой, удобно использовать пакеты для разделения модели на составные части, после чего объединить диаграммы классов и деятельности, нарисовав пару диаграмм состояний для тех классов, которые имеют интересные жизненные циклы.

Построенную подобным образом исходную модель предметной области следует рассматривать как базовый остов, а вовсе не как некоторую модель высокого уровня. Термин «модель высокого уровня» означает отсутствие множества деталей. В нескольких ситуациях я уже встречался с подобной ошибкой, состоящей, например, в установке «Не указывайте в данных моделях атрибуты». В результате получаются совершенно бессодержательные модели. Вполне очевидно, что такие модели вызывают насмешки у разработчиков.

Однако вы не можете воспользоваться и прямо противоположным подходом и строить детальную модель. Если вы пойдете по этому пути, потребуются столетия, и вы умрете от «аналитического» паралича. Выход заключается в выявлении наиболее важных деталей и концентрации усилий именно на них. Большинство деталей выявится в ходе итеративной разработки. Именно поэтому я предпочитаю рассматривать подобную модель как базовый остов, который служит фундаментом для последующих моделей. Хотя она и содержит детали, но составляет только небольшую часть нашего проекта.

Естественно, она не скажет вам, как отделить «мясо от костей»; именно в этом состоит искусство опытного аналитика, а я пока не разобрался, как это делается.

Как только определены варианты использования, они позволят управлять дальнейшим моделированием предметной области. При появлении очередных вариантов использования команда разработчиков должна оценить, содержат ли они нечто такое, что способно оказать сильное влияние на модель предметной области. Если содержат, то их нужно исследовать и далее, а если нет, то такие варианты использования следует отложить на некоторое время в сторону.

Команда, занятая построением модели предметной области, должна представлять собой небольшую группу (от двух до четырех человек), в состав которой входят разработчики и эксперты предметной области. Наименее жизнеспособная команда состоит из одного разработчика и одного эксперта предметной области.

Эта команда должна интенсивно работать в течение всей фазы исследования, пока не завершатся все прения относительно этой модели. В течение этого времени руководитель проекта должен следить за тем, чтобы команда, с одной стороны, не завязла в трясине мелких деталей,

а с другой стороны, не ушла в рассмотрение операций слишком высокого уровня. И то и другое не позволит им твердо стоять ногами на земле. Когда они обретут понимание того, что делают, погружение в детали станет для них самой большой опасностью. В этом случае задание жесткого срока завершения работы поможет сконцентрировать внимание в нужном направлении.

Для понимания требований к системе следует построить прототип для любых составных частей вариантов использования. Построение прототипа (прототипирование, prototyping) – это хороший способ для достижения наилучшего понимания динамики функционирования системы.

Я использую построение прототипа всякий раз, когда у меня нет полной уверенности относительно того, каким образом будет функционировать подверженная риску часть системы. Такого прототипа может оказаться вполне достаточно для понимания степени риска и оценки объема работы по его снижению. Обычно я не строю прототип для картины в целом, а напротив, использую общую модель предметной области, чтобы выделить те ее части, которые требуют построения прототипов.

Думаю, что разработчикам, впервые приступающим к использованию языка UML, построение прототипов более необходимо. Это поможет им понять, как диаграммы языка UML соответствуют современному программированию.

Когда вы используете некоторый прототип, не следует ограничиваться только той средой, в которой выполняется разработка конечного продукта. Например, для построения и анализа прототипа я часто использую язык Smalltalk, даже если разрабатываю программную систему на языке C++.

Одним из наиболее важных элементов, который следует принимать во внимание при анализе рассматриваемого риска, является обеспечение доступа к знаниям экспертов в предметной области. Отсутствие доступа к лицам, действительно знающим предметную область, служит повсеместно причиной провала проектов. Следует пойти на значительные затраты времени и финансовых средств для привлечения в вашу команду лиц, по-настоящему знающих предметную область – качество программного обеспечения прямо пропорционально знаниям этих экспертов. Нет необходимости использовать их все время, но следует постоянно стремиться к новым знаниям, к более глубокому пониманию и с готовностью обсуждать любые вопросы.

Технологические риски

Самый хороший способ справиться с технологическими рисками – это попытаться построить прототипы на основе той технологии, которую вы предполагаете использовать.

Предположим, например, что вы собираетесь использовать язык C++ и реляционную базу данных. Вам необходимо построить простое приложение, используя совместно язык C++ и базу данных. Попробуйте взять для этого несколько средств и оценить, какое из них работает наилучшим образом. Потратьте некоторое время, чтобы выяснить, удобно ли для работы то средство, которое вы собираетесь использовать.

Не следует забывать, что наибольшие технологические риски возникают в процессе согласования компонентов в едином проекте, а вовсе не присутствуют в каждом из компонентов в отдельности. Вы можете хорошо разбираться как в языке C++, так и в реляционных базах данных, однако их совместное применение может принести вам неприятные сюрпризы. Именно поэтому так важно на ранней стадии процесса разработки иметь в наличии все компоненты, которые вы намереваетесь использовать в дальнейшем, и согласовать их друг с другом.

На этой стадии также следует рассмотреть каждое из решений по архитектуре проекта. Обычно они принимают форму идей по составу основных компонентов и способу их построения. Это особенно важно, если вы намереваетесь строить распределенную систему.

Как часть этого исследования, следует сосредоточить внимание на тех аспектах, которые по вашему мнению будет трудно изменить впоследствии. Старайтесь разрабатывать свой проект таким образом, который позволил бы вам относительно легко вносить изменения в элементы данного проекта. Задайте себе следующие вопросы:

- Что случится, если какая-то часть технологии не будет работать?
- Что произойдет, если не удастся согласовать между собой два фрагмента мозаики?
- Какова вероятность принятия ошибочных решений? Если они все-таки будут иметь место, каким образом мы могли бы с ними справиться?

Так же как и модель предметной области, вам следует внимательно анализировать варианты использования по мере их появления, чтобы оценить, нет ли в них чего-либо такого, что может привести ваш проект к краху. Если есть опасения, что варианты использования могут содержать «скрытого червя», продолжайте их исследовать и далее.

В ходе этого процесса, как правило, приходится пользоваться рядом методов языка UML для того, чтобы схематически представить идеи и документировать те решения, которые впоследствии вы постараетесь реализовать. Не пытайтесь на этой стадии разработать исчерпывающую модель; краткие наброски – это все, что необходимо, и все, что следует использовать.

- Диаграммы классов (см. главы 4 и 6) и диаграммы взаимодействия (см. главу 5) полезно использовать для представления взаимодействия компонентов.

- Диаграммы пакетов (см. главу 7) на этой стадии могут дать общее представление о компонентах системы.
- Диаграммы развертывания (см. главу 10) могут дать представление о распределении составных частей системы.

Риски, связанные с квалификацией персонала

Я часто посещаю различные конференции и с интересом слушаю выступления разработчиков, только что завершивших какой-либо объектно-ориентированный проект. В ответ на вопрос: «Что являлось для вас самой большой ошибкой?» они почти всегда включают следующую фразу: «Нам следовало больше внимания уделять обучению».

Я никогда не перестану удивляться тому, как компании приступают к выполнению сложных объектно-ориентированных проектов, имея незначительный опыт в этой области и не задумываясь о наиболее выгодных решениях. Людям жаль тратить деньги на обучение, но им влетает в копеечку продление сроков завершения проекта.

Обучение является способом избежать ошибок, поскольку учителя эти ошибки уже сделали. Исправление ошибок требует времени, а время стоит денег. Таким образом, вы все равно заплатите ту же сумму, только без необходимых знаний и опыта выполнение проекта продлится гораздо дольше.

Я не являюсь большим сторонником формальных курсов обучения, хотя сам обучался на многих курсах и даже разработал несколько из них. Однако я так и не смог убедиться в том, что с помощью этих курсов можно овладеть мастерством объектно-ориентированной технологии. Они дают людям общее представление о том, что они должны знать, но не позволяют обрести квалификацию, необходимую для выполнения того или иного серьезного проекта. Небольшой курс обучения может быть полезным, но это всего лишь начало.

Если вы намерены пройти небольшой курс обучения, уделите самое серьезное внимание выбору преподавателя. Стоит потратить больше средств на такого преподавателя, который обладает соответствующими знаниями и способен их излагать в занимательной форме. При этом организуйте свое обучение в форме непродолжительных занятий и только в то время, когда в этом возникает необходимость. Если вы не примените на практике полученные знания сразу после окончания курса, то скоро их забудете.

Наилучший способ научиться хорошо применять объектно-ориентированные методы – обратиться за помощью к **опытному наставнику**, под руководством которого вы смогли бы работать над проектом в течение достаточного периода времени. Такой человек покажет, как делать те или иные вещи, будет наблюдать за тем, как продвигается ваша работа, и давать по ходу дела полезные советы и рекомендации.

Наставник вникнет в специфику данного проекта и разберется, какие практические приемы лучше использовать. На ранних стадиях проекта он является одним из участников команды, помогая принимать те или иные решения. Со временем вы станете более опытными, и наставник станет больше наблюдать, чем делать сам. Моя цель как наставника состоит в том, чтобы мое присутствие в команде перестало быть необходимым.

Вы можете привлечь таких опытных людей для работы как по отдельным направлениям, так и для всего проекта в целом. Они могут быть заняты в проекте в течение всего времени его выполнения или только его части. Многие наставники предпочитают работать по одной неделе над каждым проектом в течение каждого месяца; другие считают, что этого слишком мало. Постарайтесь найти такого наставника, который не только обладает знаниями, но и может передать их другим. Хороший наставник может оказаться наиболее важным фактором успеха вашего проекта; не экономьте на качестве.

Если у вас нет возможности привлечь для работы знающего помощника, организуйте обсуждение проекта каждую пару месяцев или около этого. За несколько дней до такого обсуждения следует пригласить опытного наставника для анализа различных аспектов проекта. В течение этого времени он сможет критически проанализировать все части вашего проекта, предложить дополнительные идеи и порекомендовать использовать какие-либо полезные методы, о которых команда может даже не подозревать. Хотя постоянный наставник может принести гораздо больше пользы, такой способ тоже может оказаться эффективным при обнаружении ключевых аспектов, которые позволят улучшить вашу работу.

Читая литературу, вы также можете повысить свою квалификацию. Старайтесь читать как минимум по одной серьезной технической книге каждый месяц. Еще лучше обсудить такую книгу с другими разработчиками. Найдите пару других разработчиков, желающих прочесть эту же книгу. Договоритесь с ними читать по несколько глав в неделю и в течение часа или двух обсуждать эти главы друг с другом. Если вы поступите таким образом, то сможете достичь лучшего понимания книги, чем при ее чтении в одиночку. Если вы являетесь менеджером, поощряйте такой подход. Предоставьте такой группе помещение и время для обсуждения, а также выделите своим сотрудникам средства на приобретение технической литературы.

В среде специалистов по образцам такая групповая работа над книгами считается особенно полезной. Уже появилось несколько групп по обсуждению образцов. Дополнительную информацию о таких группах можно получить в Интернете по адресу: *http://www.hillside.net/patterns*.

На стадии исследования следует обращать внимание на любые аспекты, по которым у вас отсутствуют знания или опыт. Планируйте приобрести нужный опыт к тому моменту, когда вам он станет необходим.

Политические риски

Я не могу предложить по этому поводу никаких серьезных советов, поскольку не слишком искушен в корпоративной политике. Могу лишь настоятельно порекомендовать вам найти компетентного в этой области эксперта.

Когда исследование заканчивается?

Согласно моему эмпирическому правилу фаза исследования должна занимать около одной пятой времени от общей продолжительности проекта. Основными признаками завершения фазы исследования являются следующие два события:

- Разработчики могут с уверенностью оценить, что следует делать в ближайшие дни и сколько времени при этом потребуется на реализацию каждого варианта использования.
- Идентифицированы все наиболее серьезные риски, а самые важные из них поняты настолько, что вам известно, как с ними справиться.

Планирование фазы построения

Существует множество способов планирования итеративного проекта. Важно понимать, что план разрабатывается с целью обеспечить осведомленность всей команды о ходе выполнения проекта. Используемый мною подход к планированию основан на методах *Экстремального программирования*, Бек (Beck), 2000 [2].

Сущность формирования плана заключается в установлении последовательности итераций построения и в определении функциональности, которую следует реализовать на каждой итерации. Некоторые разработчики предпочитают работать с небольшими вариантами использования и на каждой итерации завершать работу с одним из них. Другие предпочитают работать с большими по масштабу вариантами использования и на отдельной итерации рассматривать только один из сценариев, а другие – на последующих итерациях. Базовый процесс при этом является тем же самым. Итак, опишем этот процесс применительно к небольшим вариантам использования.

В ходе планирования я предпочитаю рассматривать две группы лиц: клиенты и разработчики.

Клиентами являются лица, которые предполагают использовать систему, не выходя за пределы внутрифирменной разработки. Для гото-

вой системы представителями клиента являются менеджеры. Главная особенность здесь заключается в том, что клиентами являются лица, которые могут оказывать влияние на бизнес-процессы в том или ином варианте использования, который подлежит реализации.

Разработчиками являются лица, которые участвуют в построении системы. Они должны адекватно оценивать затраты и объемы работ, необходимые для реализации отдельного варианта использования. Я считаю, что оценку должны выполнять именно разработчики, а не менеджеры. При этом нужно быть уверенным, что разработчик, оценивающий данный вариант использования, разбирается в этом наилучшим образом.

Первый шаг состоит в классификации вариантов использования. Это можно сделать двумя способами.

Во-первых, клиент делит варианты использования на три части в соответствии с их бизнес-значимостью: высокие, средние и низкие. (Заметим, что здесь следует придерживаться по возможности более точных оценок.) Затем клиент расписывает содержимое каждой категории.

После этого разработчики упорядочивают варианты использования в соответствии с риском разработки. Например, «высокую степень риска» следует использовать для чего-либо, что представляется очень трудным для выполнения, может привести к неудаче проекта или до сих пор не является хорошо понятным.

После того как это сделано, разработчики должны оценить продолжительность времени в человеко-неделях, которое потребуется для реализации каждого варианта использования. При выполнении такой оценки учитывайте время, необходимое для анализа, проектирования, кодирования, тестирования модулей, их интеграции и подготовки документации. При этом следует придерживаться принципа, что все разработчики полностью согласны с решениями друг друга без влияния деструктивных аспектов. (Мы рассмотрим дополнительный фактор принятия спорных решений позже.)

Подготовив такие оценки в тот или иной момент времени, можно сделать вывод о том, в состоянии ли вы выполнить намеченный план или нет. Проанализируйте варианты использования с высокой степенью риска. Если большая часть времени работы над проектом тратится на эти варианты использования, то вам необходимо выполнить дополнительное исследование.

Следующий шаг состоит в определении периода времени для вашей итерации. Вы можете установить фиксированную продолжительность итерации, чтобы иметь возможность получать результаты с заданной регулярностью. Для вариантов использования, которые доставляют много хлопот, следует увеличить продолжительность соответствующей итерации. Например, при использовании языка Smalltalk она мо-

жет продолжаться от двух до трех недель, а языка C++ – от шести до восьми недель.

Теперь мы можем рассмотреть трудоемкость каждой итерации.

Отметим, что соответствующие оценки будут сделаны в предположении, что вся команда работает в согласии друг с другом. Очевидно, это далеко не всегда так. Поэтому для других случаев следует ввести поправочный коэффициент, который способен учесть разницу между идеальным и реальным временем. Этот коэффициент может быть получен путем сравнительных оценок выполнения реальных проектов.

Теперь следует оценить скорость вашей работы над проектом. Другими словами, оценить объем работ, который вы можете выполнить в течение некоторой итерации. Его можно рассчитать, зная количество разработчиков в команде, умножив его на продолжительность итерации и разделив результат на поправочный коэффициент. Например, пусть имеется 8 разработчиков, длительность итерации составляет 3 недели, а поправочный коэффициент равен 2. В этом случае трудоемкость отдельной итерации в идеале составит 12 человеко-недель ($8 \times 3 \times 1/2$).

Просуммируйте время, необходимое для реализации всех вариантов использования, разделите на трудоемкость одной итерации и добавьте на всякий случай единицу. В результате получится исходная оценка количества итераций, которое потребуется для выполнения проекта.

Следующий шаг заключается в распределении вариантов использования по итерациям.

Варианты использования, которые обладают высоким приоритетом и/или риском разработки, следует реализовывать в первую очередь. *Нельзя* откладывать рассмотрение риска в последнюю очередь! При этом может возникнуть необходимость разделить слишком большие варианты использования и, возможно, пересмотреть предварительные оценки некоторых вариантов использования в соответствии с порядком их реализации. Реально может потребоваться меньше усилий, чем трудоемкость итерации, но никогда не следует планировать больше, чем позволяют ваши способности.

Для оценки внедрения выделите 10–35% от времени построения на тонкую настройку и конфигурирование конечного продукта. (Если у вас нет опыта выполнения этих операций в конкретной обстановке, то выделите еще больше времени.)

Затем добавьте коэффициент учета непредвиденных обстоятельств: от 10 до 20% времени построения в зависимости от степени оцениваемого вами риска. Прибавьте этот коэффициент ко времени окончания фазы внедрения. Для своей команды следует планировать поставку конечного продукта без учета этих непредвиденных обстоятельств, однако приступать к поставке конечного продукта следует после окончания непредвиденного времени.

После применения всех перечисленных рекомендаций у вас будет некоторый **план проекта**, в котором для каждой итерации указаны варианты использования, подлежащие реализации в ходе ее выполнения. Этот план символизирует согласие между разработчиками и пользователями. Такой план вовсе не следует считать чем-то застывшим – разумеется, всякий должен иметь возможность внести необходимые изменения в план по ходу выполнения проекта. Однако поскольку этот план отражает согласие между разработчиками и пользователями, изменения в него должны вноситься совместно.

Таким образом, как можно видеть из предыдущего обсуждения, варианты использования служат основой для планирования проекта, и именно поэтому в языке UML им уделяется такое серьезное внимание.

Построение

Построение системы выполняется путем последовательности итераций. Каждая итерация представляет собой в некотором смысле мини-проект. На каждой итерации для соответствующих ей вариантов использования должны быть выполнены анализ, проектирование, кодирование, тестирование и интеграция. Итерация завершается демонстрацией результатов пользователям и тестированием системы, которое проводится с целью контроля правильности реализации вариантов использования.

Цель этого процесса состоит в снижении степени риска. Часто причиной возникновения риска служит откладывание решения сложных проблем на самый конец проекта. Мне доводилось встречать проекты, в которых тестирование и интеграция выполнялись в самую последнюю очередь. Тестирование и интеграция – это достаточно масштабные задачи, и они всегда занимают больше времени, чем предполагают разработчики. Выполняя их в последнюю очередь, можно столкнуться с серьезными проблемами и внести в проект нежелательную дезорганизацию. Именно поэтому я всегда поощряю моих клиентов разрабатывать самотестируемое программное обеспечение (см. врезку).

Итерации на стадии построения являются как инкрементными (наращиваемыми), так и повторяющимися.

- Итерации являются *инкрементными* в смысле некоторой функции. Каждая итерация реализует очередные варианты использования и добавляет их к уже реализованным в ходе предыдущих итераций.
- Итерации являются *повторяющимися* в смысле разрабатываемого кода. На каждой итерации некоторая часть существующего кода переписывается заново с целью сделать его более гибким.

Самотестируемое программное обеспечение

С возрастом я все серьезнее отношусь к тестированию. Тестирование должно быть непрерывным процессом. Не следует писать программный код до тех пор, пока вы не знаете, как его тестировать. Написав программный код, сразу же пишите для него тесты. Пока не будет выполнено тестирование, нельзя утверждать, что вы завершили написание кода.

Однажды написанный тестовый код должен храниться вечно. Разрабатывайте тестовый код таким образом, чтобы вы смогли запускать каждый тест с помощью простой командной строки или нажатия кнопки на графическом интерфейсе пользователя (GUI). Результатом выполнения теста должно быть сообщение «ОК» или список ошибок. Кроме того, все тесты должны проверять свои собственные результаты. Ничто не приводит к более неоправданной трате времени, как получение на выходе теста числового значения, понимание смысла которого требует дополнительных усилий.

Я выполняю как модульное, так и функциональное тестирование. Модульные тесты следует писать самим разработчикам. После чего они организуются на пакетной основе и кодируются с целью тестирования интерфейсов всех классов. По моему мнению, написание модульных тестов, как это ни удивительно, увеличивает мою скорость программирования.

Функциональные или системные тесты должны разрабатываться отдельной небольшой командой, работа которой состоит только в тестировании. Такая команда должна рассматривать всю систему как черный ящик, а нахождение ошибок должно доставлять ее участникам особое удовольствие. (Зловещие усы и нахальные усмешки не обязательны, но желательны.)

Для модульного тестирования существует простая, но довольно мощная основа с открытым кодом: семейство xUnit. Более подробную информацию по этой теме можно найти на моей домашней странице в Интернете.

В последнем случае очень полезным может оказаться метод **реорганизации** (см. врезку). Хорошо бы обратить внимание на то, какой объем кода оказывается ненужным после каждой итерации. Если каждый раз выбрасывается менее 10% предыдущего кода, то это должно вызывать подозрение.

Интеграция должна представлять собой непрерывный процесс. Для участников проекта каждая итерация должна заканчиваться полной интеграцией соответствующих результатов. Однако интеграция может

Реорганизация

Можете ли вы интерпретировать принцип энтропии применительно к программному обеспечению? Этот принцип предполагает, что программы в самом начале являются хорошо спроектированными, но по мере добавления в них какой-либо новой функциональности постепенно утрачивают свою структуру, превращаясь в конечном счете в бесформенную массу спагетти.

Отчасти это происходит по причине масштаба. Предположим, вы написали небольшую программу, которая хорошо выполняет конкретную работу. Пользователи просят вас расширить функциональность программы, в результате чего она становится все более сложной. Как бы тщательно ни отслеживался проект, подобная ситуация все равно может произойти.

Одной из причин возрастания энтропии является следующее обстоятельство. Когда вы добавляете в программу новую функцию, то, как правило, дописываете новый код поверх существующего и совсем не заботитесь о том, как это может повлиять на прежнюю программу. В такой ситуации можно было бы либо перепроектировать существующую программу, чтобы внести изменения наилучшим образом, либо попытаться учесть возможность внесения изменений в выполняемом проекте.

Хотя теоретически программу лучше перепроектировать, обычно это приводит к большой дополнительной работе, поскольку любое переписывание существующей программы порождает новые ошибки и проблемы. Вспомните старую инженерную поговорку: «Пока не сломалось – не отлаживай». Однако если вы не будете перепроектировать свою программу, внесение дополнений может оказаться более сложным, чем могло бы быть в случае перепроектирования.

Со временем такая сверхсложность приведет к чрезмерным дополнительным затратам. Поэтому приходится идти на компромисс: хотя перепроектирование и сопряжено с серьезными трудностями в краткосрочном плане, зато приносит выгоду в долгосрочной перспективе. Находясь под давлением жестких плановых сроков, большинство разработчиков предпочитает откладывать эти трудности на будущее.

Реорганизация представляет собой термин, используемый для описания методов, которые позволяют уменьшить краткосрочные трудности, связанные с перепроектированием. В процессе реорганизации вы изменяете не функциональность своей программы, а лишь ее внутреннюю структуру, для того чтобы сделать ее более понятной и приспособленной к изменениям.

Реорганизационные изменения обычно касаются небольших действий: переименование метода, перемещение атрибута из одного класса в другой, объединение двух подобных методов в суперклассе. Каждое такое действие очень мало, однако даже пары часов, затраченных на внесение таких небольших изменений, может оказаться достаточно для существенного улучшения программы.

Реорганизацию можно упростить, если следовать следующим принципам:

- Не следует одновременно заниматься реорганизацией программы и расширением ее функциональности. Необходимо проводить четкую грань между этими действиями. В процессе работы можно на короткое время переключаться от выполнения одного действия к другому – например, в течение получаса заниматься реорганизацией, затем в течение часа работать над добавлением новой функции и снова полчаса потратить на реорганизацию только что добавленного кода.
- До начала реорганизации убедитесь в том, что у вас имеются хорошие тесты.
- Выполняйте только небольшие и тщательно продуманные действия. Переместите некоторый атрибут из одного класса в другой. Объедините два подобных метода в один суперкласс. Протестируйте программу после каждого такого действия. Хотя вы не получите сразу впечатляющих результатов, подобные действия позволят избежать дополнительной отладки и, следовательно, сократят общее время разработки.

Реорганизацию следует выполнять всякий раз при добавлении новой функции или обнаружении ошибки. Не следует выделять на реорганизацию специальное временя; лучше делать это понемногу, но каждый день.

Дополнительную информацию по реорганизации можно найти в книге Фаулера (Fowler),1999 [19].

и должна выполняться еще чаще. Хорошим правилом является ежедневное выполнение полной сборки и интеграции. Если это делать каждый день, не потребуются дополнительные усилия для последующей синхронизации компонентов. А вот если интеграция отложена на поздние сроки, могут возникнуть серьезные проблемы.

Разработчику следует выполнять интеграцию всякий раз после завершения каждого значительного объема работы. Кроме того, для гарантии требуемого уровня тестирования на каждой итерации следует запускать полный набор тестов для всех модулей.

Когда план заканчивается неудачей

Чтобы быть уверенным в выполнении плана, вам необходимо знать только одну вещь, а именно – все что происходит, происходит в соответствии с планом. Управление планом состоит главным образом в эффективном отслеживании происходящих изменений.

Важной чертой итеративной разработки является фиксация временных сроков – вам не позволят пропустить ту или иную дату. Тем не менее, в результате обсуждения и по соглашению с клиентом реализация вариантов использования может быть перенесена на более позднюю итерацию. Главное здесь в том, чтобы приобрести привычку регулярно отслеживать даты и избавиться от плохой привычки их забывать.

Если вы сами обнаружили задержку в реализации большого количества вариантов использования, то самое время пересмотреть план, включая переоценку уровня трудоемкости вариантов использования. На этой стадии разработчики должны предложить более конструктивные идеи относительно продолжительности отдельных этапов. Вам следует *преднамеренно* вносить изменения в план через каждые две или три итерации.

Использование языка UML на фазе построения

На этой стадии могут оказаться полезными все методы языка UML. Поскольку я собираюсь говорить о методах, рассмотреть которые у меня еще не было возможности, можете спокойно пропустить этот раздел и вернуться к нему позже.

Если вы намерены добавить очередной вариант использования, то в первую очередь применяйте его для уточнения сферы вашей деятельности. Для предварительного описания некоторых понятий варианта использования и определения того, как эти понятия согласуются с уже разработанным программным обеспечением, может оказаться полезной концептуальная диаграмма классов (см. главу 4).

Преимущество этих методов на данной стадии заключается в том, что они могут быть использованы в сотрудничестве с экспертом предметной области. Как говорит Брэд Кэйн (Brad Kain): «Анализ возможен только тогда, когда рядом находится эксперт предметной области (в противном случае это не анализ, а псевдоанализ)».

В ходе выполнения проекта постарайтесь понять, как будут кооперироваться классы с целью обеспечения функциональности, требуемой каждым вариантом использования. Я полагаю, что для представления подобных взаимосвязей могут оказаться весьма полезными CRC-карточки и диаграммы взаимодействия. При этом будут выявлены ответственности и операции, которые могут быть представлены на диаграмме классов.

Рассматривайте эти результаты как предварительный набросок и как средство для обсуждения с вашими коллегами различных подходов к проектированию. Только достигнув согласия в этих вопросах, можно приступать к написанию кода.

Не прощающий ничего программный код неизбежно вскроет все недостатки проекта. Не бойтесь вносить изменения в проект в процессе его изучения. Если подобные изменения серьезны, то следует использовать специальные обозначения для последующего обсуждения идей со своими коллегами.

После того как вы разработали программное обеспечение, для подготовки документации о проделанной работе может быть использован язык UML. Я пришел к выводу, что диаграммы языка UML оказываются весьма полезными для достижения общего понимания системы. Однако должен подчеркнуть, что при этом вовсе не имею в виду построение детальных диаграмм для системы в целом. В этой связи уместно процитировать Уорда Каннингхема (Ward Cunningham), 1996 [16]:

> *Тщательно подобранные и аккуратно написанные заметки могут легко заменить традиционную обширную проектную документацию. Последняя скорее затмевает суть, за исключением некоторых отдельно взятых аспектов. Выделите эти аспекты ... и забудьте обо всем остальном.*

Я полагаю, что подробную документацию следует разрабатывать на основе готового программного кода (подобно, например, JavaDoc). Чтобы обратить внимание на наиболее важные понятия, лучше подготовить дополнительную документацию. Считайте ее составной частью первого этапа знакомства с программой, перед тем как приступить к детальному изучению кода. Я предпочитаю представлять документацию в форме текста, достаточно краткого, чтобы прочесть его за чашкой кофе, и с использованием диаграмм языка UML, которые помогают придать наглядный характер обсуждению.

Я использую диаграмму пакетов (см. главу 7) как своего рода логическую карту дорог системы. Эта диаграмма помогает мне понять логические блоки системы и обнаружить зависимости между ними (держа их под контролем). Диаграмма развертывания (см. главу 10) может существенно улучшить понимание на этой стадии, поскольку позволяет представить общую физическую картину системы.

Для каждого пакета предпочтительнее строить диаграмму классов уровня спецификации. При этом я не указываю каждую операцию в том или ином классе, а показываю только те ассоциации, ключевые атрибуты и операции, которые помогают мне понять основные идеи реализации. Такая диаграмма классов служит своего рода оглавлением в форме графической таблицы.

Если некоторый класс в течение своего жизненного цикла имеет сложное поведение, то для описания такого поведения я изображаю диаграмму состояния (см. главу 8). Но поступаю так только в том случае,

когда поведение действительно является достаточно сложным, что, кстати, встречается не так уж и часто. Обычно сложными бывают взаимодействия между классами, для описания которых я строю диаграммы взаимодействия.

В книгу также включаются некоторые фрагменты программного кода, достаточно важные для системы и написанные аккуратным стилем. Для особо сложных алгоритмов используется диаграмма деятельности (см. главу 9), но только в том случае, если она помогает понять алгоритм лучше, чем сам код.

Если я сталкиваюсь с часто повторяющимися понятиями, то для описания их основных идей использую образцы (см. врезку).

Образцы

Язык UML позволяет описать объектно-ориентированный проект. С другой стороны, **образцы** фиксируют внимание на результатах этого процесса – на моделях примеров.

Многие аналитики высказывают мнение о том, что целый ряд проблем при выполнении проектов вызван весьма слабым представлением разработчиков об отдельных проектных решениях, которые хорошо известны более опытным коллегам. Образцы как раз и служат для описания наиболее общих способов решения проблем. Они накапливаются у тех разработчиков, которые умеют находить в своих проектах повторяющиеся решения. Эти разработчики описывают каждое решение таким образом, чтобы другие люди могли изучить этот образец и понять, как его применить.

Рассмотрим следующий пример. Допустим, имеется несколько объектов, которые выполняются некоторым процессом на вашем персональном компьютере, при этом им необходимо взаимодействовать с другими объектами, которые выполняются в рамках другого процесса. Возможно, второй процесс также выполняется на вашем компьютере, а может быть, и где-нибудь еще. При этом вы хотите избавить объекты вашей системы от проблем, связанных с поиском других объектов в сети или с выполнением вызовов удаленных процедур.

Для реализации такого пожелания можно создать внутри вашего локального процесса некий *объект-заместитель* (proxy) для удаленного объекта. Заместитель обладает тем же интерфейсом, что и удаленный объект. Ваши локальные объекты будут обращаться к заместителю, используя обычный механизм передачи сообщений внутри процесса. При этом заместитель отвечает за последующую передачу любых сообщений реальному объекту, где бы тот ни находился.

На рис. 2.2 изображена некоторая диаграмма классов (см. главу 4), которая иллюстрирует структуру образца *Заместитель*.

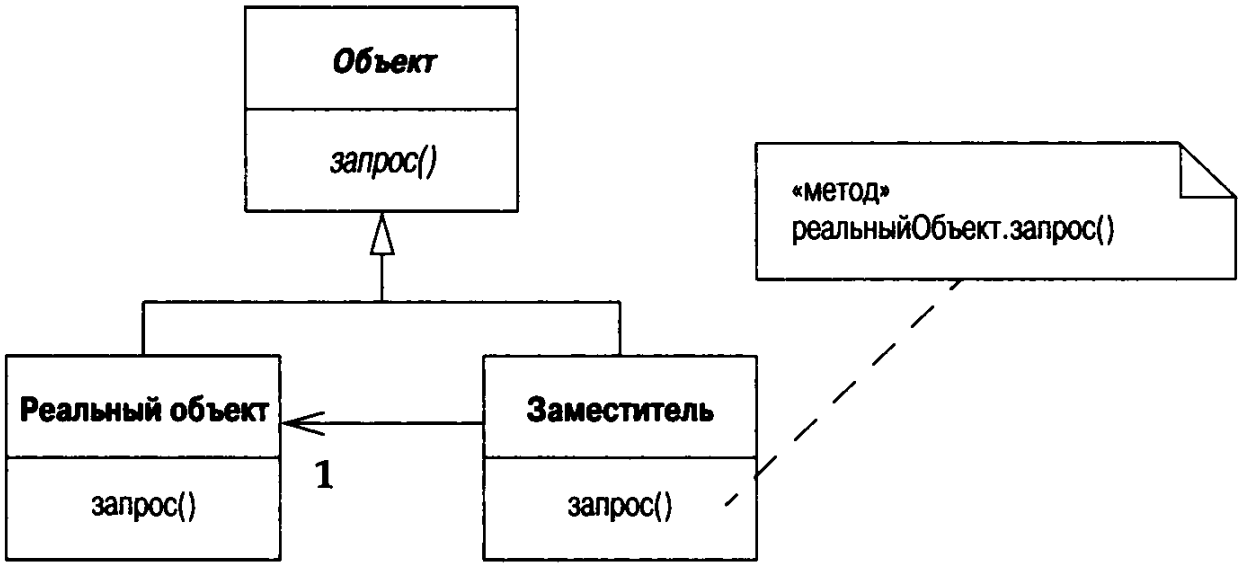


Рис. 2.2. Структура проектного образца Заместитель

Заместители являются обычным средством, используемым в вычислительных сетях и других приложениях. Разработчиками уже накоплен большой опыт в использовании заместителей, в частности, как их можно применять, какие при этом можно получить преимущества, какими обладают ограничениями и как их следует реализовать. В книгах по методологии, подобных этой, такой опыт не обсуждается; все они рассматривают только способы графического изображения заместителя. Хотя все это также полезно, тем не менее, обсудить опыт использования заместителей было бы более интересно.

Начиная с 1990-х годов, некоторые разработчики стали обобщать свой опыт в этом направлении. Они объединились в сообщество, заинтересованное в написании образцов. Эти разработчики выступили в качестве спонсоров конференций и выпустили ряд книг.

Наиболее известной из этих книг является книга «Банды четырех» (Гамма, Хелм, Джонсон и Влиссидес, 1995 [20]), в которой детально рассматриваются 23 проектных образца.

Если вы хотите узнать что-либо о заместителях, именно эта книга является самым подходящим источником. В ней с десяток страниц посвящен каждому образцу, при этом детально описывается взаимодействие объектов, преимущества и недостатки этого образца, его модификации и особенности использования, а также даются полезные советы по его реализации на языках Smalltalk и C++.

Заместитель – это проектный образец, поскольку он описывает некоторый метод проектирования. Образцы также могут существовать и в других областях. Предположим, вы проектируете систему для управления рисками на финансовых рынках. Вам необходимо знать, как с течением времени изменяется стоимость портфеля ценных бумаг. Это можно сделать, отслеживая цену каждой акции через определенные промежутки времени и отмечая ее изменение. Но вам также желательно иметь возможность оценивать степень риска в различных гипотетических ситуациях (например, «Что произойдет, если резко упадет цена на нефть?»).

Для этого можно построить некий *Сценарий*, который содержит все множество цен на акции. После чего можно определить отдельные *Сценарии* для цен за прошедшую неделю, оптимистических прогнозов на следующую неделю, прогнозов на следующую неделю в предположении резкого падения цен на нефть и т. д. Образец *Сценарий* (рис. 2.3) является образцом анализа, поскольку он описывает фрагмент модели предметной области.

Рис. 2.3. Образец Сценария для анализа

Предназначенные для анализа образцы имеют большое значение, поскольку именно с них лучше всего начинать работу в новой предметной области. Я сам начал коллекционировать образцы для анализа, когда столкнулся с серьезными проблемами в некоторых новых предметных областях. Я знал, что являлся отнюдь не первым разработчиком, кто занимается их моделированием, но всякий раз был вынужден начинать с чистого листа бумаги.

Интересной особенностью образцов для анализа является то, что они могут оказаться полезными в самый неожиданный момент. Когда я начал работать над проектом корпоративной системы финансового анализа, мне чрезвычайно помогли образцы, накопленные мною ранее при выполнении проекта для системы здравоохранения.

Дополнительную информацию об образце *Сценарий* или других образцах анализа можно найти в моей книге (Фаулер, 1997 [18]).

Образец – это нечто гораздо большее, чем просто модель. Образец должен также включать обоснование, почему он именно такой, какой есть. Часто говорят, что образец является решением той или иной проблемы. Образец должен придать проблеме необходимую ясность, объяснить, почему он является решением данной проблемы, а также в каких ситуациях он работает или не работает.

Образцы очень важны, поскольку являются этапом, следующим за пониманием основ языка или метода моделирования. Образцы предлагают вам набор готовых решений, а также показывают, что делает модель хорошей и как ее построить. Они обучают на примерах.

Когда я только начал заниматься проектированием, то постоянно удивлялся, почему мне все время приходится начинать на пустом месте. Почему бы не иметь под рукой справочники, которые показали бы мне, как делать самые общие вещи? Сообщество разработчиков образцов как раз и пытается создать такие справочники.

Когда следует использовать образцы

Всякий раз при попытке что-либо разработать, будь то анализ, проектирование, кодирование или управление проектом, следует поискать какие-либо подходящие образцы, которые могли бы вам помочь.

Где найти дополнительную информацию

Упомянутые ранее книги служат прекрасным введением в образцы. Дополнительную информацию можно найти в Интернете на сайте, посвященном образцам, по адресу: *http://www.hillside.net/patterns*. Именно здесь можно познакомиться с самой современной информацией о состоянии мира образцов.

Внедрение

Цель итеративной разработки заключается в том, чтобы сделать весь процесс разработки более последовательным, в результате чего команда разработчиков смогла бы получить готовый программный продукт. Однако есть некоторые вещи, которые не следует выполнять слишком рано. Первой среди них является оптимизация.

Хотя оптимизация и повышает производительность системы, но уменьшает ее прозрачность и расширяемость. Именно здесь необходимо принять компромиссное решение – в конце концов, система должна быть достаточно производительной, чтобы удовлетворять требованиям пользователей. Слишком ранняя оптимизация затруднит последующую разработку, поэтому ее следует выполнять в последнюю очередь.

На стадии внедрения не следует дополнять конечный продукт новой функциональностью, кроме, может быть, самой минимальной и абсолютно необходимой. Именно на этой фазе *следует* выявлять ошибки. Хорошим примером фазы внедрения может служить период между выпуском бета-версии и появлением окончательной версии продукта.

Когда использовать итеративную разработку

Итеративную разработку следует использовать только в тех проектах, в которых вы желаете добиться успеха.

Может быть, это звучит несколько поверхностно, но с годами я становлюсь все большим сторонником использования итеративной разработки. При грамотном применении она является весьма важным методом, который может быть использован для раннего выявления риска и достижения лучшего управления процессом разработки. Однако это не означает, что можно вовсе обойтись без управления проектом (хотя, если быть справедливым, я должен отметить, что некоторые используют ее именно для этой цели). Итеративная разработка требует тщательного планирования. Это весьма серьезный подход, и поэтому любая книга по объектно-ориентированной разработке рекомендует его использовать.

Где найти дополнительную информацию

Имеется довольно много специальной литературы, посвященной рассмотрению процесса. Я отдаю предпочтение двум книгам:

- Кокбёрн (Cockburn), 1998 [12] проделал прекрасную работу, рассмотрев ключевые аспекты в столь небольшой книге. Именно поэтому я рекомендую ее для первоначального знакомства с управлением объектно-ориентированными проектами.
- Мак-Коннелл (McConnell), 1996 [31] представил глубокий анализ наилучших практических методов.

Что касается Рационального унифицированного процесса, то дополнительная информация содержится в:

- книге Крухтена (Kruchten), 1999 [27], которая представляет собой краткое изложение данной темы.
- книге Джекобсона, Буча и Рамбо, 1999 [23], где процесс описан более детально.

Если вас интересуют вопросы нового и еще развивающегося подхода, познакомьтесь с книгой Кента Бека (Kent Beck), 2000 [2] по экстремальному программированию. Этот подход существенно отличается от рассматриваемого, поскольку уделяет основное внимание тестированию и развитию проекта. См. также по адресу: *http://www.armaties.com/exsreme.htm*.

3

Варианты использования

Варианты использования представляют собой интересный феномен. Долгое время как в процессе объектно-ориентированной, так и традиционной разработки аналитики использовали типовые сценарии, которые помогали им лучше понять требования к системе. Однако эти сценарии трактовались довольно неформально – постоянно используя, их редко документировали. Свою известность Айвар Джекобсон (Ivar Jacobson) получил благодаря тому, что разработанный им метод Objectory и посвященная этому методу книга [24] изменили эту ситуацию.

Расширив содержание вариантов использования, А. Джекобсон повысил их значимость, что позволило превратить варианты использования в основной элемент разработки и планирования проекта. Со времени публикации его книги (1992) объектное сообщество в значительной степени одобрило применение вариантов использования.

Что же такое вариант использования?

Прямого ответа на этот вопрос не существует. Но попытаться на него ответить можно, описав вначале сценарий.

Сценарий представляет собой последовательность шагов, описывающих взаимодействие между пользователем и системой. Таким образом, если мы рассмотрим реализованный на веб-технологии интернет-магазин, то можно представить следующий сценарий покупки товаров в этом магазине:

> *Покупатель просматривает каталог и помещает выбранные товары в корзину. При желании оплатить покупку он вводит информа-*

цию о кредитной карточке и совершает платеж. Система проверяет авторизацию кредитной карточки и подтверждает оплату товара тотчас же и по электронной почте.

Подобный сценарий описывает только одну ситуацию, которая может иметь место. Если авторизация кредитной карточки окажется неудачной, то подобная ситуация может послужить предметом уже другого сценария.

В таком случае **вариант использования** представляет собой множество сценариев, объединенных вместе некоторой общей целью пользователя. В нашем случае вы можете построить вариант использования «Покупка товара», который охватывает оба сценария – как успешной оплаты, так и неудачной авторизации. Для вариантов использования могут быть и другие альтернативные пути продолжения сценариев. Часто вы можете столкнуться с тем, что вариант использования представляет самую общую ситуацию, которая включает множество альтернатив как заканчивающихся неудачей, так и приводящих к успешному завершению.

Ниже представлен простой формат для записи варианта использования, в котором исходный сценарий описан в виде последовательности нумерованных шагов, а альтернативы могут изменять эту последовательность (рис. 3.1).

Покупка товара

1. Покупатель просматривает каталог и выбирает товары для покупки.
2. Покупатель оценивает стоимость всех товаров.
3. Покупатель вводит информацию, необходимую для доставки товара (адрес, доставка на следующий день или в течение трех дней).
4. Система предоставляет полную информацию о цене товара и его доставке.
5. Покупатель вводит информацию о кредитной карточке.
6. Система осуществляет авторизацию счета покупателя.
7. Система выполняет немедленную оплату товаров.
8. Система подтверждает оплату товаров для покупателя по адресу его электронной почты.

Альтернатива: *Неудача авторизации*

На шаге 6 система получает отрицательный ответ на запрос о состоянии счета покупателя.

Необходимо предоставить покупателю возможность повторно ввести информацию о кредитной карточке и выполнить ее авторизацию.

Альтернатива: *Постоянный покупатель*

3а. Система предоставляет информацию о текущей покупке и ее цене, а также последние 4 цифры информации о кредитной карточке.

3б. Покупатель может согласиться или отказаться от предложенной системой информации.

После этого перейти на шаг 6 исходного сценария.

***Рис. 3.1.** Текст примера варианта использования*

Существует множество способов записи содержания вариантов использования; язык UML в этом смысле не определяет никакого стан-

дарта. При этом вы можете добавить в вариант использования дополнительные секции. Например, можно ввести дополнительную секцию для предусловий, выполнение которых является обязательным для того, чтобы началась реализация отдельного варианта использования. Просмотрите несколько книг, в которых рассматриваются варианты использования, и дополните свой вариант теми элементами, которые имеют для вас значение. Однако не включайте в вариант ничего лишнего, что не сможет оказать вам реальную помощь.

Приведем важный пример того, как могут уточняться варианты использования. Рассмотрим другой сценарий покупки в интернет-магазине, при котором покупатель уже известен системе как постоянный покупатель. Некоторые аналитики будут рассматривать эту ситуацию как третий сценарий, в то время как другие выделят ее в отдельный вариант использования. Вы можете также применить одно из отношений между вариантами использования, которые будут описаны позже.

Количество деталей в сценарии зависит от риска в соответствующем варианте использования: чем больше риск, тем больше деталей необходимо указать. Часто случается так, что на фазе исследования я детально описываю только небольшое количество вариантов использования, в то время как остальные из них содержат не больше информации, чем вариант использования на рис. 3.1. В процессе итерации вы можете добавить в вариант использования больше деталей, если они необходимы для его реализации. При этом можно не записывать все детали явным образом; часто очень эффективно их вербальное понимание.

Диаграммы вариантов использования

Когда А. Джекобсон в 1994 г. [24] предложил варианты использования в качестве основных элементов процесса разработки программного обеспечения, он ввел также диаграмму для их наглядного представления. **Диаграмма вариантов использования** в настоящее время также является частью языка UML.

Многие аналитики находят такую диаграмму полезной. Однако нет необходимости рисовать эту диаграмму при описании вариантов использования. В одном из наиболее удачных из известных мне проектов каждый вариант использования записывался на отдельной помеченной карточке, которые впоследствии раскладывались по пачкам, чтобы показать, что необходимо выполнять на каждой итерации.

На рис. 3.2 показаны некоторые варианты использования для финансовой торговой системы (трейдинг).

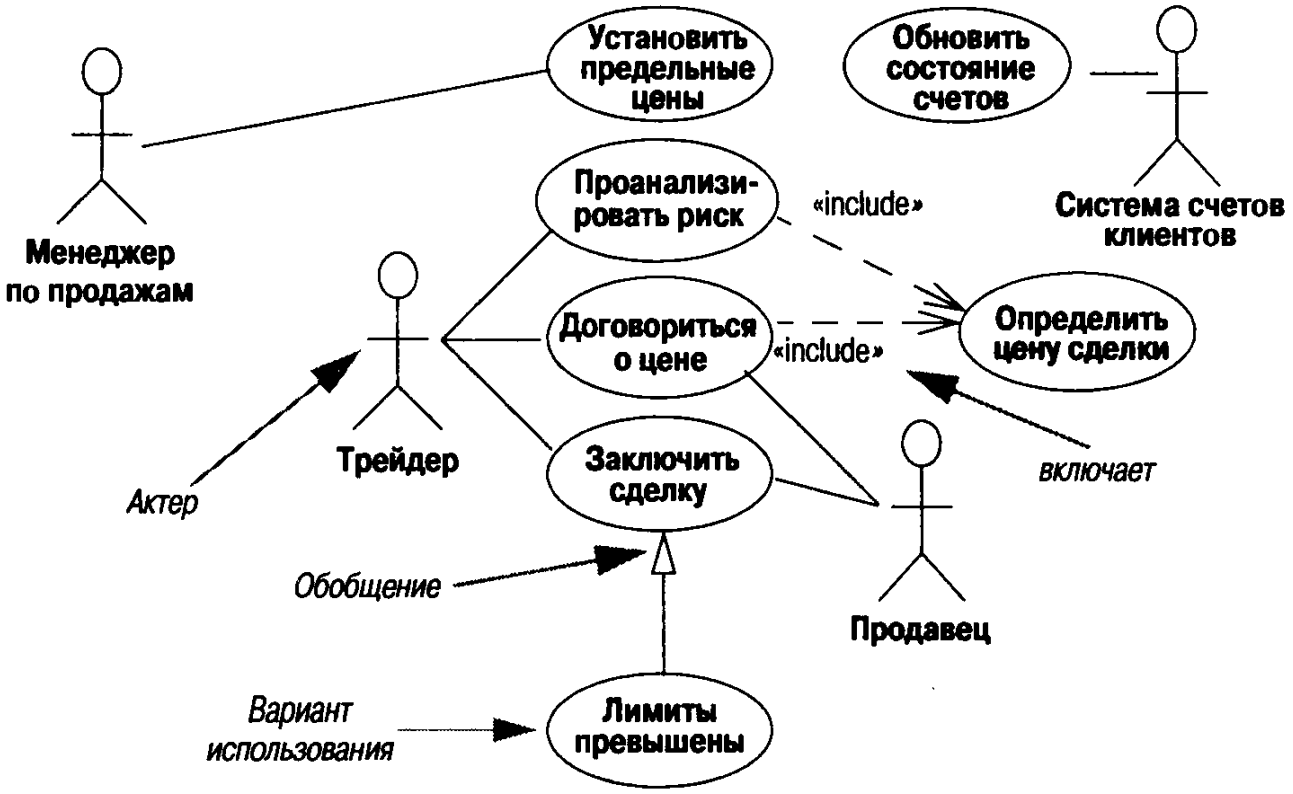


Рис. 3.2. Диаграмма вариантов использования

Актеры

Актер представляет собой некоторую роль, которую играет пользователь по отношению к системе. На рис. 3.2 представлены 4 актера: менеджер по продажам, трейдер (оптовый торговец), продавец и система счетов клиентов. (Да, я знаю, что было бы лучше использовать слово «роль», но, по всей видимости, имел место неточный перевод со шведского языка.)

Вероятно, в конкретной организации будет много трейдеров, но все они по отношению к системе играют одну и ту же роль. Отдельный пользователь может играть и более одной роли. Например, один из старших трейдеров может играть роль менеджера по продажам и являться при этом постоянным трейдером. Трейдер может быть также продавцом. Когда имеешь дело с актерами, важно думать о ролях, а не о людях или их работе.

Актеры связаны с вариантами использования. Один актер может выполнять несколько вариантов использования; в свою очередь, у варианта использования может быть несколько актеров, которые его выполняют.

Я пришел к выводу, что на практике актеры наиболее полезны при попытке сформулировать варианты использования. В случае большой системы часто трудно определить список вариантов использования. В подобной ситуации бывает проще сначала перечислить всех актеров,

после чего для каждого из них попытаться разработать варианты использования.

Актеры не обязаны быть людьми, хотя на диаграмме вариантов использования они изображаются в виде стилизованных человеческих фигурок. Отдельный актер может быть даже внешней системой, которой необходима некоторая информация от разрабатываемой системы. На рис. 3.2 мы можем видеть подобную ситуацию при необходимости обновления состояния счетов для системы счетов клиентов.

Существует несколько вариантов изображения актеров. Некоторые разработчики на диаграмме вариантов использования указывают каждую внешнюю систему или пользователя как актера; другие предпочитают указывать только инициатора варианта использования. Я предпочитаю изображать актера, который получает от варианта использования некоторую информацию или сервис, при этом некоторые разработчики считают таких актеров первичными.

Однако я далек от подобной точки зрения. В связи с этим вынужден заметить, что хотя система счетов клиентов получает сервис от разрабатываемой системы, на диаграмме вариантов использования не показывается тот факт, что кто-либо из актеров может получать информацию от самой системы счетов клиентов. Эту ситуацию следует представить при разработке модели собственно системы счетов клиентов. Другими словами, вам следует всегда подвергать сомнению взаимосвязи вариантов использования с актерами системы, выясняя истинные цели пользователей и рассматривая альтернативные пути достижения этих целей.

В процессе работы с актерами и вариантами использования я не очень беспокоился о том, насколько точно установлены отношения между ними. Большую часть времени я посвящал вариантам использования; актеры же являлись лишь средством для их получения. Пока не выявлялись все варианты использования, меня не интересовало детальное описание актеров.

Иногда все же встречаются ситуации, когда, возможно, следует больше внимания уделять актерам.

- Для различных видов пользователей может потребоваться переконфигурация системы. В этом случае каждый вид пользователя представляет собой актера, а варианты использования укажут вам, что должен делать каждый из этих актеров.
- Акцент на получении желаемого сервиса от вариантов использования может помочь вам установить приоритеты между различными актерами.

Некоторые варианты использования могут не иметь очевидных связей с отдельными актерами. Рассмотрим, например, некоторое коммунальное предприятие. Очевидно, одним из его вариантов использования является: отправить счет за пользование коммунальными услуга-

ми. Однако при этом совсем непросто идентифицировать соответствующего актера. Нет такой специфической роли пользователя, которая запрашивала бы этот счет. Счет направляется потребителю коммунальных услуг, но сам потребитель не будет возражать, если этого не произойдет. В этой ситуации наиболее подходящим актером может стать бухгалтерия предприятия, поскольку именно она пользуется результатами этого варианта использования. Тем не менее, обычно полагают, что бухгалтерия не имеет непосредственного отношения к рассматриваемому варианту использования.

Следует отметить, что некоторые варианты использования могут не проявиться даже в результате долгих размышлений над вариантами использования для каждого из актеров. Если это произойдет, то не стоит особенно беспокоиться. Гораздо важнее хорошо разбираться в имеющихся вариантах использования и тех сервисах, которые они предоставляют пользователям.

Хорошим источником идентификации вариантов использования являются внешние события. Поразмышляйте обо всех событиях внешнего мира, которые вы способны осмыслить. То или иное событие может оказывать влияние на систему даже без участия пользователей или, наоборот, получать воздействия, главным образом, от пользователей. Идентификация событий, на которые необходимо реагировать, поможет вам идентифицировать и варианты использования.

Отношения между вариантами использования

Помимо связей между актерами и вариантами использования, на диаграммах могут быть представлены также отношения между вариантами использования.

Отношение **включения** (include) имеет место, когда имеется какой-либо фрагмент поведения системы, который повторяется более чем в одном варианте использования, и вы не хотели бы, чтобы его описание копировалось в каждом из этих вариантов использования. Например, для обоих вариантов использования «Проанализировать риск» и «Договориться о цене» необходимо определить цену сделки. Описание определения цены сделки можно представить как отдельный сценарий, после чего вставлять его в нужное место. Поэтому я выделил отдельный вариант использования «Определить цену сделки» и ссылаюсь на него из других вариантов использования.

Если имеется один вариант использования, который подобен другому варианту использования, но намного шире его, то такое отношение может быть представлено как **обобщение вариантов использования** (use case generalization). По существу, это дает нам другой способ построения альтернативных сценариев.

В нашем примере основным вариантом использования является вариант «Заключить сделку». Предполагается, что все складывается бла-

гополучно. Однако некоторые обстоятельства могут помешать совершению сделки. Одним из них является превышение лимитов, например, максимальной суммы торговой сделки, установленной для отдельного клиента. В этом случае будет нарушен обычный ход выполнения процесса, ассоциированного с данным вариантом использования, и необходимо предусмотреть его изменение.

Это изменение можно учесть в рамках основного варианта использования «Заключить сделку» как альтернативу, подобно варианту использования «Покупка товара», который был рассмотрен ранее. Однако нас не покидает ощущение, что эта альтернатива не настолько самостоятельна, чтобы выделить ее в отдельный вариант использования. Поэтому поместим альтернативную ветвь процесса в специальный вариант использования, который ссылается на базовый вариант использования. Этот специальный вариант использования может замещать любую часть базового варианта использования, тем не менее, он по-прежнему должен быть ориентирован на выполнение важных для пользователя действий.

Третье отношение, которое изображено на рис. 3.2, называется **расширением** (extend). В сущности, оно аналогично обобщению, но имеет некоторые дополнительные особенности.

При построении модели расширяющий вариант использования может дополнять поведение базового варианта использования, но в базовом варианте должны быть определены так называемые «точки расширения». При этом расширяющий вариант использования может дополнять поведение базового только в этих точках расширения (рис. 3.3).

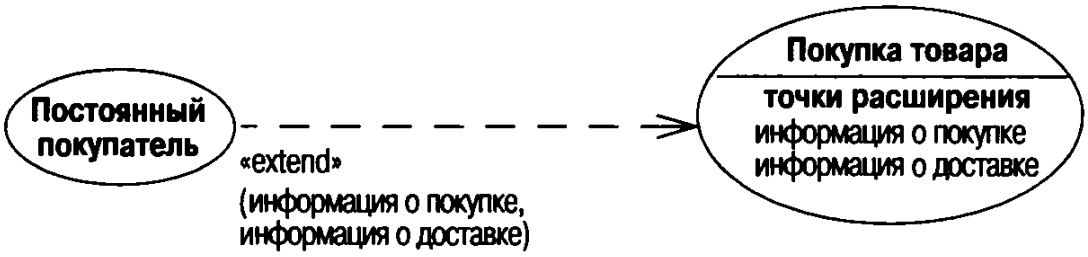


Рис. 3.3. Отношение расширения

Вариант использования может иметь несколько точек расширения, и расширяющий вариант использования может расширять базовый в одной или нескольких точках расширения. Вы должны указать на диаграмме стереотип «extend» на линии, которая соединяет соответствующие варианты использования.

Как обобщение, так и расширение позволяют выполнять расщепление или декомпозицию вариантов использования. На этапе исследования я часто расщепляю те из вариантов использования, которые представляются мне слишком сложными. Я также расщепляю их на этапе построения проекта, если у меня создается впечатление, что я не могу реализовать такой вариант использования в течение одной итерации це-

ликом. Когда я провожу такую декомпозицию, то вначале предпочитаю рассматривать обычную ситуацию и лишь затем – ее вариации.

В связи с этим можно воспользоваться следующими правилами:

- Используйте отношение *включения*, когда приходится повторять одно и тоже в двух и более отдельных вариантах использования и есть желание исключить это повторение.
- Используйте отношение *обобщения*, когда описываете изменение некоторого нормального поведения и есть желание сделать это поверхностно.
- Используйте отношение *расширения*, когда описываете изменение некоторого нормального поведения и есть желание сделать это в более точной форме, определив точки расширения в базовом варианте использования.

Варианты использования бизнес-процессов и систем

Общей проблемой при работе с вариантами использования является такая ситуация, когда, уделяя основное внимание взаимодействию пользователя с системой, можно упустить из рассмотрения тот факт, что лучшим способом решения проблемы может оказаться изменение бизнес-процесса.

Часто можно услышать разговоры раэработчиков о вариантах использования систем и вариантах использования бизнес-процессов. Конечно, эта терминология не является точной, но обычно считается, что вариант использования системы описывает особенности взаимодействия с программным обеспечением, в то время как вариант использования бизнес-процесса представляет собой реакцию на действие клиента или некоторое событие.

У меня нет особых причин углубляться в рассмотрение этих вопросов. На ранних этапах исследования я больше изучаю варианты использования бизнес-процессов, но думаю, что варианты использования системы являются более полезными для планирования. По-моему, размышления о вариантах использования бизнес-процесса приносят большую пользу, особенно при рассмотрении альтернативных способов реализации потребностей актеров.

В своей работе я вначале концентрирую внимание на вариантах использования бизнес-процесса, после чего перехожу к рассмотрению вариантов использования системы, которые должны обеспечивать выполнение этого бизнес-процесса. В конце этапа исследования я рассчитываю иметь по меньшей мере одно множество вариантов использования системы для каждого из вариантов использования бизнес-процесса. При этом, как минимум, варианты использования бизнес-процесса следует идентифицировать и стараться специфицировать их в первую очередь.

Когда следует применять варианты использования

Я просто не могу представить себе ситуацию, в которой можно было бы обойтись без вариантов использования. Они являются совершенно необходимым средством при анализе требований, планировании и управлении итеративной разработкой. Работа с вариантами использования является одной из самых важных задач на этапе исследования.

Большая часть ваших вариантов использования сформируется на этапе исследования проекта, однако в ходе дальнейшей работы вы можете обнаружить дополнительные особенности. Имейте это постоянно в виду и не ослабляйте свое внимание. Каждый вариант использования является потенциальным требованием к системе, и пока оно не выявлено, вы не сможете перейти к планированию его реализации.

Некоторые разработчики вначале составляют перечень вариантов использования и обсуждают их, после чего приступают к моделированию. По моему мнению, концептуальное моделирование с участием пользователей помогает выявить варианты использования. Поэтому я склонен одновременно работать с вариантами использования и заниматься концептуальным моделированием.

Важно помнить, что варианты использования служат *внешним* представлением системы. Именно поэтому не следует ожидать каких-либо взаимозависимостей между вариантами использования и классами внутри системы.

Сколько вариантов использования следует сформировать? В ходе недавнего заседания комиссии OOPSLA некоторые эксперты по вариантам использования утверждали, что для проекта с трудоемкостью 10 человеко-лет необходимо около дюжины вариантов использования. Но это только базовые варианты использования; каждый из таких вариантов использования может заключать в себе несколько сценариев и альтернативных вариантов использования. Мне также встречались проекты аналогичной трудоемкости с более чем сотней отдельных вариантов использования. (Если подсчитать количество альтернативных вариантов использования для дюжины базовых вариантов использования, то их общее количество как раз и будет около 100.) Сколько нужно, столько и используйте их в своей работе.

Где найти дополнительную информацию

Хорошей и небольшой книгой по вариантам использования является книга Шнайдера (Schneider) и Винтера (Winters), 1998 [39]. В этом издании рассматриваются еще отношения языка UML версии 1.1, такие как использование и расширение, но по моему мнению она остается лучшей книгой по работе с вариантами использования. Следует также

рекомендовать сборник работ Алистера Кокбёрна (Alistair Cockburn) в Интернете по адресу: *http://members.aol.com/acockburn*.

Первая книга Айвара Джекобсона, 1994 [24] послужила толчком для многих других работ, и она по-прежнему остается современной. Следующая книга А. Джекобсона, 1995 [25] также является полезной, поскольку в ней сделан акцент на вариантах использования бизнес-процессов.

4

Диаграммы классов: основы

Диаграмма классов по праву занимает центральное место в объектно-ориентированном подходе. Фактически любая методология включает в себя некоторую разновидность диаграмм классов.

Помимо своего широкого применения диаграммы классов концентрируют в себе большой диапазон понятий моделирования. Хотя их основные элементы используются практически всеми, более сложные понятия применяются не так часто. Именно поэтому я разделил рассмотрение диаграмм классов на две части: основы (данная глава) и дополнительные понятия (см. главу 6).

Диаграмма классов описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. Имеётся два основных вида статических отношений:

- **ассоциации** (например, клиент может взять напрокат ряд видеокассет);
- **подтипы** (медсестра является разновидностью личности).

На диаграммах классов изображаются также атрибуты классов, операции классов и ограничения, которые накладываются на связи между объектами.

На рис 4.1 изображена типичная диаграмма классов.

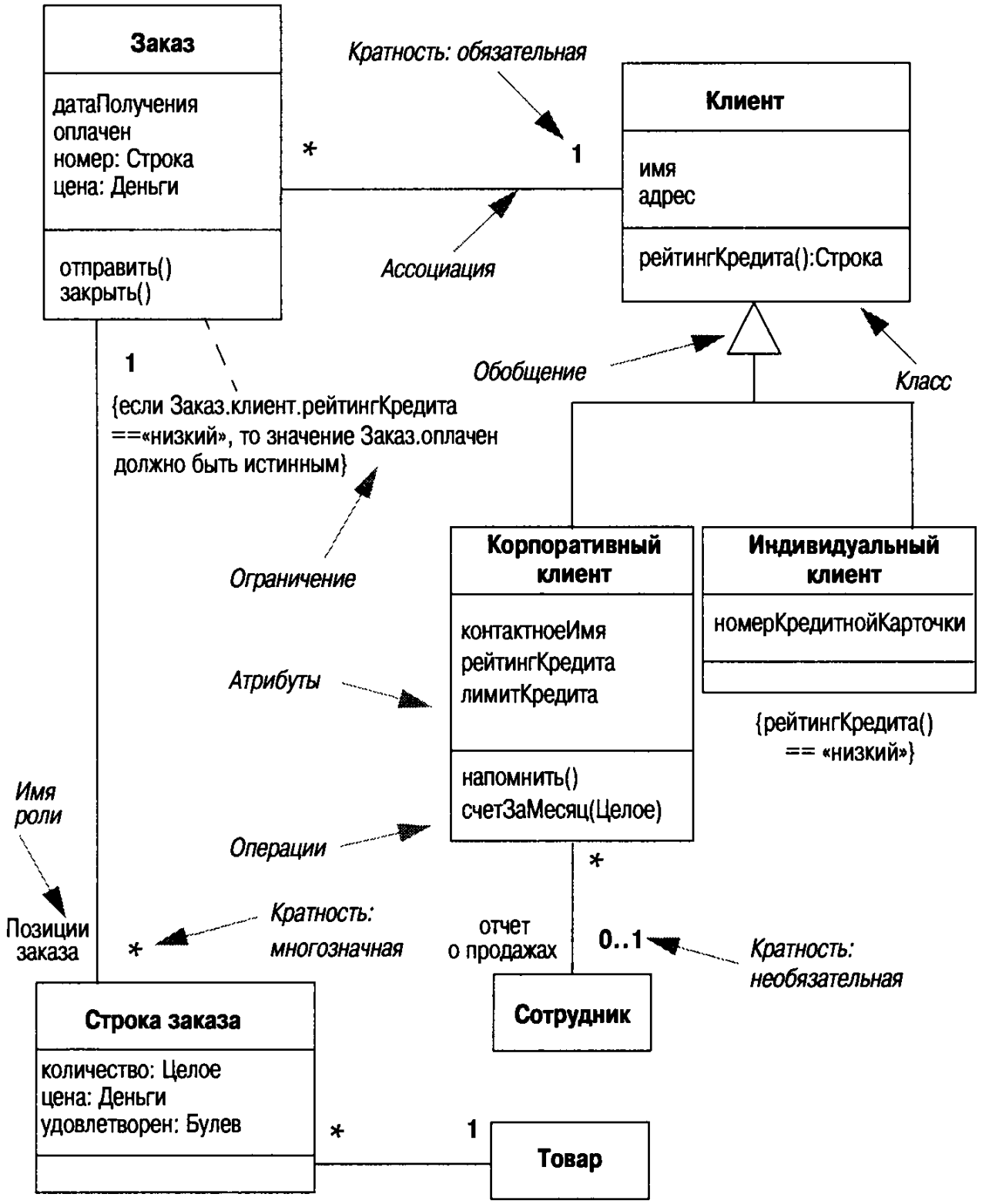


Рис. 4.1. Диаграмма классов

Особенности представления

Перед тем как приступить к описанию диаграмм классов, хотелось бы обратить ваше внимание на одну тонкость, связанную с характером использования этих диаграмм разработчиками. Эта тонкость обычно никак не документируется, однако существенно влияет на способ интерпретации диаграмм и поэтому имеет серьезное отношению к тому, что именно вы описываете с помощью модели.

Следуя сказанному Стивом Куком (Steve Cook) и Джоном Дэниелсом (John Daniels), 1994 [13], я утверждаю, что существуют три различные точки зрения на построение диаграмм классов или любой другой модели, однако эти различия в наибольшей степени касаются диаграмм классов:

- **Концептуальная точка зрения.** Если рассматривать диаграммы классов с концептуальной точки зрения, то они служат для представления понятий изучаемой предметной области. Эти понятия, естественно, будут соответствовать реализующим их классам, однако такое прямое соответствие зачастую отсутствует. В действительности, концептуальная модель может иметь весьма слабое отношение или вообще не иметь никакого отношения к реализующему ее программному обеспечению, поэтому ее можно рассматривать независимо от языка программирования. (Кук и Дэниелс называют такую точку зрения первичной).
- **Точка зрения спецификации.** В этом случае мы переходим к рассмотрению программной системы, при этом рассматриваем только ее интерфейсы, но не реализацию. Объектно-ориентированная разработка подчеркивает существенное различие между интерфейсом и реализацией, но на практике оно часто игнорируется, поскольку нотация класса в объектно-ориентированных языках программирования объединяет в себе как интерфейс, так и реализацию. Это весьма досадно, поскольку ключевым фактором эффективного объектно-ориентированного программирования является программирование именно интерфейса класса, а не его реализации. Это хорошо описано в первой главе книги Гаммы и др., 1995 [20]. Вы часто слышите слово «тип», когда речь идет об интерфейсе класса; тип может иметь несколько классов, которые его реализуют, а класс может реализовывать несколько типов.
- **Точка зрения реализации.** С этой точки зрения мы действительно имеем дело с классами, опустившись на уровень реализации. Эта точка зрения, вероятно, встречается наиболее часто, однако во многих ситуациях точка зрения спецификации является более предпочтительной для аналитика.

Понимание точки зрения разработчика крайне важно как для построения, так и для чтения диаграмм классов. К сожалению, различия между отмеченными особенностями представления не столь отчетливы, и большинство разработчиков при построении диаграмм смешивают различные точки зрения. Я полагаю, что часто не имеет большого значения различие между концептуальной точкой зрения и точкой зрения спецификации, и гораздо важнее различать особенности представления с точки зрения спецификации и реализации.

В дальнейшем при рассмотрении диаграмм классов я постараюсь подчеркнуть, насколько сильно каждый элемент диаграммы зависит от точки зрения.

Особенности представления диаграмм классов не являются частью формального описания языка UML, однако я пришел к выводу, что они являются исключительно важными при построении и анализе моделей. Язык UML можно использовать с любой из этих точек зрения. Вы можете явно указать особенность представления, снабдив класс стереотипом (см. главу 6). Можно также пометить класс как «класс реализации», чтобы явно указать на точку зрения реализации, или пометить его как «тип» для концептуальной точки зрения и точки зрения спецификации.

Ассоциации

На рис. 4.1 изображена простая диаграмма классов, понятная каждому, кто имел дело с обработкой заказов клиентов. Рассмотрим каждый фрагмент этой диаграммы и укажем их возможную интерпретацию с различных точек зрения.

Начнем с ассоциаций. **Ассоциации** представляют собой отношения между экземплярами классов (сотрудник работает в компании, компания имеет несколько офисов).

С **концептуальной** точки зрения ассоциации представляют концептуальные отношения между классами. На диаграмме показано, что Заказ может поступить только от одного Клиента, а Клиент в течение некоторого времени может сделать несколько Заказов. Каждый из этих Заказов может содержать несколько Строк заказа, причем каждая Строка заказа должна соответствовать единственному Товару.

Каждая из ассоциаций имеет два **конца ассоциации**; при этом каждый из концов ассоциации присоединяется к одному из классов этой ассоциации. Конец ассоциации может быть явно помечен некоторой меткой. Такая метка называется **именем роли**. (Концы ассоциации часто называют **ролями**.)

Так, например, на рис. 4.1 конец ассоциации, направленной от класса Заказ к классу Строка заказа, имеет название *Позиции заказа*. Если такая метка отсутствует, концу ассоциации присваивается имя класса-цели; например, конец ассоциации от класса Заказ к классу Клиент может быть назван *клиент*.

Конец ассоциации также обладает **кратностью**, которая показывает, сколько объектов может участвовать в данном отношении. На рис. 4.1 символ «*» возле класса Заказ для ассоциации между классами Заказ и Клиент показывает, что с одним клиентом может быть связано много заказов; напротив, символ «1» показывает, что каждый из заказов может поступить только от одного клиента.

В общем случае кратность указывает нижнюю и верхнюю границы количества объектов, которые могут участвовать в отношении. При этом символ «*» означает диапазон *0..бесконечность*: клиент может не сде-

лать ни одного заказа, но верхний предел количества заказов, сделанных одним клиентом, никак не ограничен (разумеется, теоретически!). 1 означает диапазон *1..1*, то есть заказ должен быть сделан одним и только одним клиентом.

На практике наиболее распространенными вариантами кратности являются «1», «*» и «0..1» (либо ноль, либо единица). В более общем случае для кратности может использоваться единственное число (например, *11* для количества игроков футбольной команды), диапазон (например, *2..4* для количества игроков карточной игры каналета) или дискретная комбинация из чисел или диапазонов (например, *2, 4* для количества дверей в автомобиле).

С точки зрения **спецификации** ассоциации представляют собой ответственности классов.

На рис. 4.1 предполагается, что существует один или более методов, связанных с классом Клиент, с помощью которых можно узнать, какие заказы сделал конкретный клиент. Аналогично в классе Заказ существуют методы, с помощью которых можно узнать, какой Клиент сделал конкретный Заказ и какие Строки заказа входят в этот Заказ.

Если установить стандартные соглашения по наименованию методов запросов, то, вероятно, я смог бы вывести из диаграммы названия этих методов. Например, можно принять соглашение, в соответствии с которым взаимно однозначные отношения реализуются посредством метода, который возвращает связанный объект, а многозначные отношения реализуются посредством итератора, указывающего на совокупность связанных объектов.

Следуя подобному соглашению, в языке Java, например, я могу определить следующий интерфейс для класса Заказ:

```
class Order {
     public Customer getCustomer();
     public Set getOrderLines();
     . . .
```

(Order – Заказ, Customer – Клиент, OrderLines – Строки заказа)

Очевидно, соглашения по программированию могут изменяться от случая к случаю и не затрагивать каждый метод, однако они могут оказаться весьма полезными с точки зрения нахождения вашего собственного стиля разработки.

Ассоциация несет также определенную ответственность за обновление соответствующего отношения. Так, например, должен существовать некоторый способ связи класса Заказ с классом Клиент. Детали этого способа на диаграмме не показаны; можно было бы включить спецификацию класса Клиент в конструктор для класса Заказ. Или это может быть метод *добавитьЗаказ*, ассоциированный с классом Клиент. Как мы увидим позже, такой способ связи можно сделать более явным посредством добавления операций в блок класса на диаграмме.

Однако эта ответственность *не* распространяется на структуру данных. Рассматривая диаграмму с точки зрения спецификации, я не могу высказать никаких предположений относительно структуры данных для классов. Я не могу сказать, да мне и не следует знать, содержит ли в действительности класс Заказ указатель на класс Клиент или же класс Заказ реализует свою функциональность посредством выполнения некоторого программного запроса к каждому экземпляру класса Клиент, чтобы выяснить, связан ли он с данным классом Заказ. Диаграмма описывает только интерфейс – и ничего более.

Если же модель рассматривается с точки зрения **реализации**, можно исходить из предположения, что между связанными классами существуют указатели в обоих направлениях. В этом случае диаграмма может нам сказать, что класс Заказ содержит поле, представляющее собой совокупность указателей на класс Строка заказа, а также указатель на класс Клиент. На языке Java мы можем выразить эту ситуацию следующим образом:

```
class Order {
     private Customer _customer;
     private Set _orderLines;
class Customer {
     private Set _orders;
```

В этом случае большинство аналитиков предполагают, что с таким же успехом могут выполняться все доступные операции, однако можно быть уверенным в этом, только взглянув на операции класса.

Теперь посмотрим на рис. 4.2. В основном он совпадает с рис. 4.1, за исключением того, что я добавил стрелки к ассоциациям. Эти стрелки показывают направление **навигации**.

В модели спецификации таким способом можно показать, что Заказ обязан ответить на вопрос, к какому Клиенту он относится, а Клиенту нет необходимости отвечать, к какому Заказу он имеет отношение. Вместо симметричных ответственностей мы указываем теперь только односторонние. На диаграмме реализации это будет обозначать, что класс Заказ содержит указатель на класс Клиент, но класс Клиент не имеет указателей на класс Заказ.

Как можно увидеть, навигация имеет существенное значение на диаграммах спецификации и реализации. Однако я не думаю, что она сохранит свою полезность при построении концептуальных диаграмм.

На концептуальных диаграммах, которые строятся в самом начале разработки, направления навигации чаще всего отсутствуют. Они появляются при построении диаграмм классов уровня спецификации и реализации. Отметим также, что направления навигации, вероятно, могут отличаться с точки зрения спецификации и реализации.

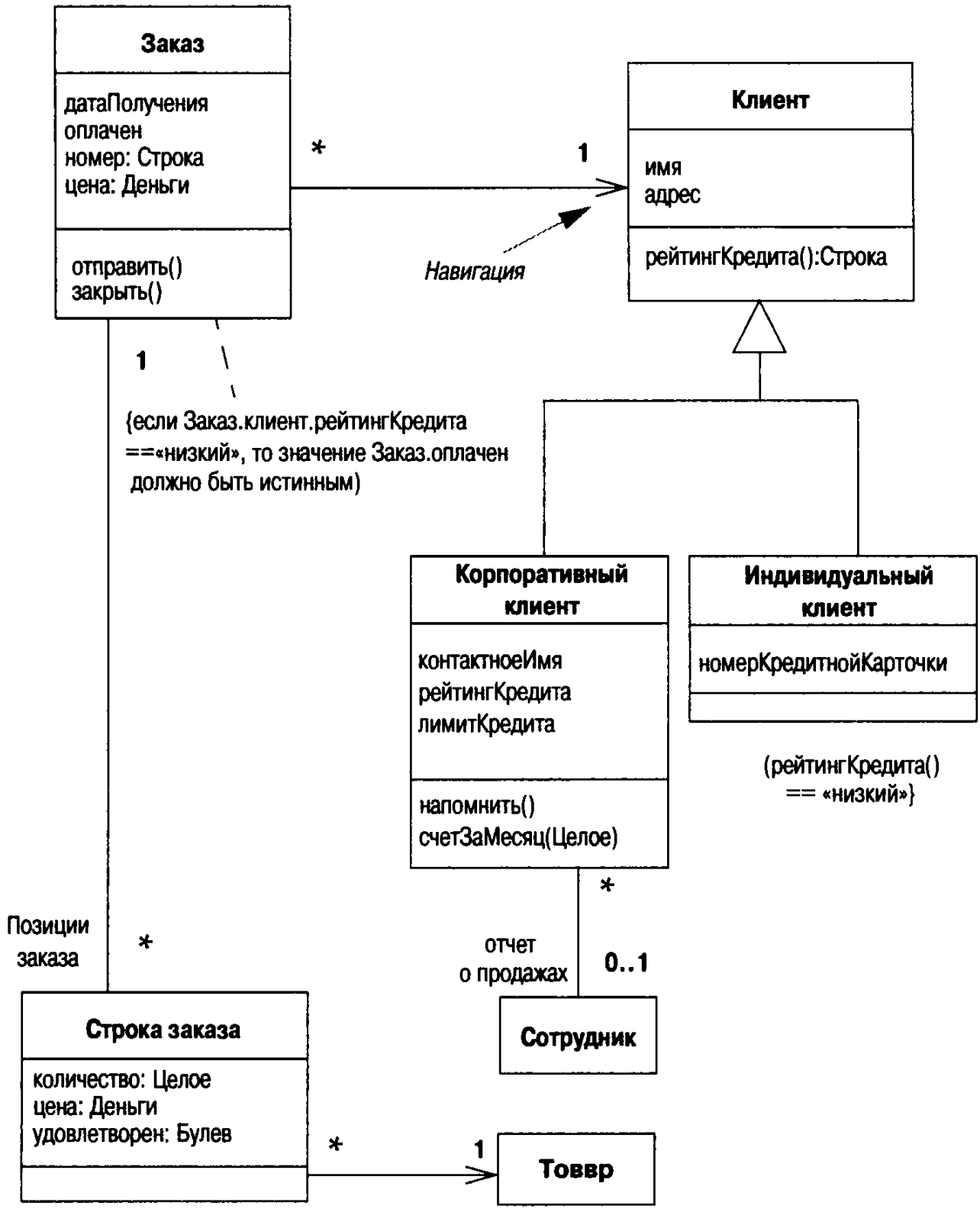


Рис. 4.2. Диаграмма классов с навигацией

Если навигация указана только в одном направлении, то такая ассоциация называется **однонаправленной ассоциацией. У двунаправленной ассоциации** навигация указывается в обоих направлениях. Если ассоциация не имеет стрелок навигации, то язык UML трактует это следующим образом: направление навигации неизвестно или ассоциация является двунаправленной. В своем проекте вы можете остановиться на любой из этих трактовок. Сам я предпочитаю трактовать от-

сутствие стрелок в моделях спецификации и реализации как неопределенное направление навигации.

Двунаправленные ассоциации содержат дополнительное ограничение, которое заключается в том, что эти две навигации являются инверсными (обратными) по отношению друг к другу. Это аналогично обозначению обратных функций в математике. Применительно к рис. 4.2 это означает, что каждая Позиция заказа, связанная с некоторым Заказом, должна быть связана с конкретным исходным Заказом. Аналогично, если взять какую-либо Строку заказа и взглянуть на Позиции заказа, соответствующие связанному с ними Заказу, то в совокупности этих позиций можно обнаружить Строку исходного заказа. Это свойство остается справедливым для любой из трех точек зрения.

Существует несколько различных способов задания имен ассоциаций. Традиционный способ именования ассоциаций, принятый у специалистов по моделированию данных, заключается в использовании фразы с глаголом, так чтобы соответствующее отношение можно было бы использовать в предложении. Большинство специалистов по объектному моделированию предпочитают использовать существительные в качестве имен для роли одного или нескольких концов ассоциации, поскольку такой способ лучше согласуется с ответственностями и операциями.

Некоторые разработчики каждой ассоциации присваивают имя. Я присваиваю ассоциации имя только в том случае, когда это улучшает понимание диаграммы. Мне приходилось видеть довольно много ассоциаций с именами типа «имеет» или «связан с». Если у конца ассоциации отсутствует имя, то я считаю, что оно совпадает с именем соответствующего класса-цели.

Ассоциация представляет перманентную связь между двумя объектами. Другими словами, такая связь существует в течение всего жизненного цикла объектов, даже если соединяемые ею экземпляры могут изменяться во времени (или при необязательной ассоциации ее вообще может не быть). Так, например, параметрическая ссылка или создание объекта вовсе не подразумевают какую-либо ассоциацию; эти элементы следует моделировать с помощью зависимостей (см. главы 6 и 7).

Атрибуты

Атрибуты очень похожи на ассоциации. На концептуальном уровне атрибут «имя Клиента» указывает на то, что клиенты обладают именами. На уровне спецификации этот атрибут указывает на то, что объект Клиент может сообщить вам свое имя и обладает некоторым способом для установления имени. На уровне реализации объект Клиент содержит некоторое поле для своего имени (называемое также переменной экземпляра или элементом данных).

В зависимости от степени детализации диаграммы обозначение атрибута может включать имя атрибута, тип и присваиваемое по умолча-

нию значение. В синтаксисе языка UML это выглядит следующим образом: *<видимость> <имя>: <тип> = <значение по умолчанию>*, где *видимость* имеет такой же смысл, как и для операций, описываемых в следующем разделе.

В чем же заключается различие между атрибутом и ассоциацией?

С концептуальной точки зрения никакого различия нет. Атрибут является еще одним видом нотации, который вы можете использовать, если сочтете это удобным. Атрибуты обычно имеют единственное значение. На диаграмме, как правило, не показывается, является ли атрибут обязательным или необязательным, хотя это можно сделать, указав кратность в квадратных скобках после имени атрибута, например *получениеДанных[0..1]: Данные*.

Различие проявляется на уровнях спецификации и реализации. Атрибуты предполагают единственное направление навигации – от типа к атрибуту. Кроме этого, подразумевается, что тип содержит только свою собственную копию атрибута объекта. В свою очередь, предполагают, что любой тип, используемый в качестве атрибута, обладает скорее значением, чем семантикой ссылки.

Значения и ссылочные типы будут рассмотрены несколько позже. В данный момент атрибуты лучше всего представить себе как небольшие, простые классы, такие как строки, даты, денежные суммы и значения, не являющиеся объектами, например *целое* и *вещественное*.

Операции

Операции представляют собой процессы, реализуемые некоторым классом. Существует очевидное соответствие между операциями и методами класса. На уровне спецификации операции соответствуют общедоступным методам над некоторым типом. Обычно можно не показывать такие операции, которые просто манипулируют атрибутами, поскольку они и так подразумеваются. Однако иногда возникает необходимость показать, что данный атрибут предназначен только для чтения (read-only) или является неизменным (frozen), то есть его значение никогда не изменяется. В модели реализации можно также указать защищенные и закрытые операции.

Полный синтаксис операций в языке UML выглядит следующим образом:

<видимость> <имя> (*<список-параметров>*): *<выражение-возвращающее-значение-типа>* {*<строка-свойств>*}

где

- *видимость* может принимать одно из трех значений: «+» общедоступная (public), «#» защищенная (protected) или «–» закрытая (private).
- *имя* представляет собой строку символов.

- *список-параметров* содержит разделенные запятой параметры, синтаксис которых аналогичен синтаксису атрибутов: *<направление> <имя>*: *<тип>* = *<значение по умолчанию>*. При этом дополнительным элементом является *направление*, которое применяется, чтобы показать характер использования параметра – для входа (*in*), выхода (*out*) или в обоих направлениях (*inout*). Если значение *направления* отсутствует, оно предполагается входным (*in*).
- *выражение-возвращающее-значение-типа* содержит список разделенных запятой значений типов. Большинство разработчиков использует только один тип возвращаемого значения, но допускается использование и нескольких таких типов.
- *строка-свойств* указывает значения свойств, которые применяются к данной операции.

Пример записи операции для счета клиента: + *показатьСостояние* (*дата: Дата*): *Деньги*.

В рамках концептуальной модели не следует использовать операции для спецификации интерфейса класса. Вместо этого их следует использовать для представления принципиальных ответственностей класса, возможно, с помощью пары слов, выражающих ответственность в CRC-карточках (см. врезку в главе 5).

По моему мнению, следует различать операции, изменяющие состояние класса, и операции, не делающие этого. Язык UML определяет **запрос** как некую операцию, которая получает некоторое значение от класса, не изменяя при этом состояние системы, другими словами, не производя побочных эффектов. Такую операцию можно пометить ограничением {запрос}. Операции, которые изменяют состояние, я называю **модификаторами**.

Полагаю, что запросы являются чрезвычайно полезными. Они могут выполняться в произвольном порядке, однако очень важно соблюдать последовательность выполнения модификаторов. В своей работе я избегаю значений, возвращаемых модификаторами, чтобы учесть указанное различие.

Другие термины, с которыми иногда приходится сталкиваться, – это методы извлечения значения (getting methods) и методы установки значения (setting methods). **Метод извлечения значения** возвращает некоторое значение из поля (и не делает ничего больше). **Метод установки значения** помещает некоторое значение в поле (и не делает ничего больше). За пределами класса клиент не способен определить, является ли запрос методом получения значения или является ли модификатор методом установки значений. Эта информация о методах является полностью внутренней для каждого из классов.

Существует еще одно различие между операцией и методом. **Операция** представляет собой вызов некоторой процедуры, в то время как **метод** является телом процедуры. Эти два понятия различают, когда имеют

дело с полиморфизмом. Если у вас есть супертип с тремя подтипами, каждый из которых переопределяет одну и ту же операцию супертипа, то вы имеете дело с одной операцией и четырьмя реализующими ее методами.

Обычно термины «операция» и «метод» используются как взаимозаменяемые, однако бывают ситуации, когда полезно их различать. Иногда разработчики различают их, используя термины *вызов метода* или *определение метода* (для операции) и *тело метода*.

В языках программирования на этот счет существуют свои собственные соглашения. В языке C++ операции называются *функциями-членами* (member functions), в то время как в языке Smalltalk операции называются *методами*. В языке C++ используется также термин *члены класса* для обозначения операций и методов класса. В языке UML для указания атрибута или операции применяется термин *свойство* (feature).

Обобщение

Типичный пример **обобщения** включает индивидуального и корпоративного клиентов некоторой бизнес-системы. Они обладают некоторыми различиями. Однако, несмотря на различия, у них много общего. Одинаковые свойства можно поместить в общий класс Клиент (супертип), при этом класс Индивидуальный клиент и класс Корпоративный клиент будут выступать в качестве подтипов.

Этот факт служит объектом разнообразных интерпретаций в моделях различных уровней. Например, на концептуальном уровне мы можем утверждать, что Корпоративный клиент является подтипом Клиента, если все экземпляры класса Корпоративный клиент по определению являются также экземплярами класса Клиент. Таким образом, класс Корпоративный клиент является частной разновидностью класса Клиент. Основная идея заключается в следующем: все, что нам известно о классе Клиент (ассоциации, атрибуты, операции), справедливо также и для класса Корпоративный клиент.

В модели уровня спецификации обобщение означает, что интерфейс подтипа должен включать все элементы интерфейса супертипа. Говорят, что интерфейс подтипа **согласован** с интерфейсом супертипа.

Другая сторона обобщения связана с принципом **замещения**. Можно подставить класс Корпоративный клиент в любой код, где требуется класс Клиент, и при этом все должно работать прекрасно. По существу, это означает, что если я написал код, предполагающий использование класса Клиент, то могу свободно использовать экземпляр любого подтипа класса Клиент. Класс Корпоративный клиент может реагировать на некоторые команды отличным от класса Клиент образом (используя полиморфизм), но это отличие не должно беспокоить вызывающий объект.

С точки зрения реализации обобщение связано с понятием наследования в языках программирования. Подкласс наследует все методы и поля суперкласса и может переопределять наследуемые методы.

Ключевой особенностью здесь является различие между обобщением с точки зрения спецификации (подтипы или наследование интерфейса) и обобщением с точки зрения реализации (подклассы или наследование реализации). Подкласс представляет собой один из способов реализации подтипа. Подтип можно также реализовать с использованием механизма делегирования – действительно, многие образцы, описанные в книге Гамма и др., 1995 [20], содержат два класса с одинаковыми интерфейсами, но без использования подклассов. В книге Фаулера, 1997 [18] можно найти описание других идей реализации подтипов.

Используя любую из этих форм реализации обобщения, вы всегда должны обеспечить справедливость этого обобщения на концептуальном уровне. Я пришел к выводу, что если этого не сделать, то можно столкнуться с серьезными проблемами, поскольку при последующих изменениях стабильность обобщения может нарушиться.

Иногда можно встретиться с ситуацией, когда подтип обладает таким же интерфейсом, как супертип, но реализует операции другим способом. При желании можно и не показывать подтип на диаграмме уровня спецификации. Я обычно указываю подтип, если в его существовании заинтересованы пользователи класса, и не указываю, если подтипы отличаются только особенностями внутренней реализации.

Правила ограничения

При построении диаграмм классов большая часть времени уходит на представление различных ограничений.

На рис. 4.2 показано, что Заказ может быть сделан только одним единственным Клиентом. Из этой диаграммы классов также следует, что каждая Позиция заказа рассматривается отдельно: вы можете заказать 40 каких-либо коричневых штучек, 40 голубых этих же штучек и 40 красных таких же штучек, но не 40 коричневых, голубых и красных штучек. Далее диаграмма утверждает, что Корпоративный клиент располагает кредитным лимитом, а Индивидуальный клиент – нет.

С помощью базовых конструкций ассоциации, атрибута и обобщения можно много сделать, специфицируя наиболее важные ограничения, но этими средствами невозможно записать каждое ограничение. Эти ограничения еще нужно каким-то образом отобразить, и диаграмма классов является вполне подходящим местом для этого.

Язык UML разрешает использовать для записи ограничений все что угодно. При этом необходимо лишь придерживаться правила: ограничения следует помещать в фигурные скобки ({}). Я предпочитаю пользоваться неформальной записью ограничений на естественном языке,

чтобы их было проще понимать. Язык UML также предоставляет для этой цели формальный язык объектных ограничений (OCL, Object Constraint Language), с которым можно познакомиться по книге Уормера (Warmer) и Клеппе (Kleppe), 1998 [45].

В идеальном случае правила ограничения следует реализовывать в виде утверждений на языке программирования. Это согласуется с понятием инварианта при проектировании по контракту (см. врезку).

Проектирование по контракту

Проектирование по контракту (Design by Contract) – это метод проектирования, разработанный Бертраном Мейером. Этот метод является центральным свойством языка Eiffel, который он разработал. Однако проектирование по контракту не является специфичным только для языка Eiffel; этот метод может быть использован и с любым другим языком программирования.

Главной идеей метода проектирования по контракту является понятие утверждения. **Утверждение** – это булево высказывание, которое никогда не должно принимать ложное значение и поэтому может быть ложным только в результате ошибки. Обычно утверждения проверяются только во время отладки и не проверяются в режиме выполнения. Действительно, при работе программы никогда не предполагается, что утверждения будут проверяться.

Метод проектирования по контракту использует три вида утверждений: предусловия, постусловия и инварианты.

Предусловия и постусловия применяются к операциям. **Постусловие** – это высказывание относительно того, как будет выглядеть окружающий мир после выполнения операции. Например, если мы определяем для числа операцию «извлечь квадратный корень», постусловие может принимать форму *вход* = *результат* * *результат*, где *результат* является выходом, а *вход* – исходное значение числа. Постусловие – это хороший способ выразить, что должно быть сделано, не говоря при этом, как это сделать. Другими словами, постусловия позволяют отделить интерфейс от реализации.

Предусловие – это высказывание относительно того, как должен выглядеть окружающий мир до выполнения операции. Для операции «извлечь квадратный корень» можно определить предусловие *вход* >= *0*. Такое предусловие утверждает, что применение операции «извлечь квадратный корень» для отрицательного числа является ошибочным и последствия такого применения будут неопределенными.

На первый взгляд эта идея кажется неудачной, поскольку нам придется выполнить некоторые дополнительные проверки, что-

бы убедиться в корректности выполнения операции «извлечь квадратный корень». При этом возникает важный вопрос: кто должен быть ответственным за выполнение этой проверки.

Предусловие явно устанавливает, что за подобную проверку отвечает вызывающий объект. Без такого явного задания ответственности мы можем получить либо недостаточную проверку (когда каждая из сторон предполагает, что ответственной является другая), либо чрезмерную проверку (когда она будет выполняться обеими сторонами). Чрезмерная проверка тоже плоха, поскольку она влечет за собой дублирование кода проверки, что, в свою очередь, может существенно увеличить сложность программы. Явное определение ответственности помогает уменьшить эту сложность. При этом может быть уменьшена опасность того, что вызывающий объект забудет выполнить проверку, поскольку утверждения обычно проверяются во время отладки и тестирования.

Исходя из этих определений предусловия и постусловия, мы можем дать строгое определение термина «исключение», которое имеет место, когда предусловие операции удовлетворяется, но операция не может возвратить значение, даже если ее постусловие тоже выполнено.

Инвариант представляет собой утверждение относительно класса. Например, класс Счет может иметь инвариант вида «*баланс = сумма(позиция.количество( ))*». Инвариант должен быть «всегда» истинным для всех экземпляров класса. «Всегда» в данном случае означает «всякий раз, когда объект инициирует выполнение какой бы то ни было операции».

По существу, это означает, что инвариант дополняет предусловия и постусловия, связанные со всеми общедоступными операциями отдельного класса. Значение инварианта может оказаться ложным во время выполнения некоторого метода, однако оно должно быть восстановлено к моменту начала взаимодействия с любым другим объектом.

Утверждения могут играть уникальную роль в задании подклассов.

Одна из опасностей полиморфизма заключается в том, что вы можете переопределить операции подкласса таким образом, что они станут несовместимыми с операциями суперкласса. Утверждения не позволят вам выполнить подобные действия. Инварианты и постусловия класса должны применяться ко всем подклассам. Подклассы могут усилить эти утверждения, но не могут ослабить их. С другой стороны, предусловие нельзя усилить, но можно ослабить.

На первый взгляд все это выглядит излишеством, однако является весьма важным для обеспечения динамического связывания. Вы всегда должны иметь возможность обратиться к экземпляру подкласса, как если бы он был экземпляром суперкласса (в соответствии с принципом замещения). Если подкласс усилил свое предусловие, то операция суперкласса, примененная к подклассу, может завершиться аварийно.

По существу, утверждения могут только усилить ответственность подкласса. Предусловия являются высказываниями, передающими ответственность вызывающему объекту; таким образом, ответственность класса можно усилить путем ослабления предусловия. На практике все эти средства позволяют намного лучше управлять процессом определения подклассов и помогают обеспечить корректное поведение подклассов.

В идеальном случае утверждения следует включать в код как часть определения интерфейса. Компиляторы должны обладать способностью выполнять проверку утверждений во время отладки и исключать ее в процессе штатной эксплуатации. Проверка утверждений может выполняться на различных этапах разработки. Часто именно предусловия предоставляют отличные возможности для обнаружения ошибок при минимальных затратах времени и сил.

Когда следует использовать проектирование по контракту

Проектирование по контракту оказывается весьма полезным методом при разработке понятных интерфейсов.

Утверждения поддерживаются как часть нотации только в языке Eiffel, но, к сожалению, Eiffel не является широко распространенным языком программирования. Для непосредственной поддержки таких утверждений в других языках программирования можно использовать дополнительные механизмы, хотя это и представляет определенные неудобства.

В языке UML об утверждениях сказано не слишком много, но их без труда можно использовать. Инварианты эквивалентны правилам ограничений на диаграммах классов, и их следует применять как можно чаще. Предусловия и постусловия операции следует документировать при определении операций.

Где найти дополнительную информацию

Книга Мейера (Meyer), 1997 [33] является классической работой в области объектно-ориентированного проектирования, в ней много внимания уделяется утверждениям. Ким Уолден и Жан-Марк Нирсон, 1995 [44], а также Стив Кук и Джон Дэниелс, 1994 [13] в своих книгах часто прибегают к использованию метода контрактного проектирования.

> Дополнительную информацию можно также получить в компании ISE, сотрудником которой является Бертран Мейер, в Интернете по адресу: *www.eifel.com*.

Когда использовать диаграммы классов

Диаграммы классов являются фундаментом почти всех объектно-ориентированных методов, поэтому вы будете работать с ними практически постоянно. Данная глава охватывает только основные понятия; в главе 6 рассматриваются более сложные понятия.

Трудность, связанная с использованием диаграмм классов, заключается в том, что их нотация слишком богата и в ней можно заблудиться. Приведем лишь несколько полезных советов по этому поводу.

- Не пытайтесь использовать сразу все доступные понятия. Начните с самых простых, которые рассмотрены в этой главе: классов, ассоциаций, атрибутов, обобщений и ограничений. Используйте дополнительные понятия, рассмотренные в главе 6, только тогда, когда это действительно необходимо.
- Выбор точки зрения для построения модели должен соответствовать конкретному этапу работы над проектом:
  - На этапе анализа стройте концептуальные модели.
  - Если вы работаете с программным обеспечением, сосредоточьте свое внимание на моделях спецификации.
  - Применяйте модели реализации только в тех случаях, когда иллюстрируете конкретный способ реализации.
- Не надо строить модели для всего на свете; вместо этого следует сконцентрироваться на главных аспектах. Лучше иметь мало диаграмм, которые постоянно используются в работе и отражают все внесенные изменения, чем иметь дело с большим количеством забытых и устаревших моделей.

Самая большая опасность, связанная с диаграммами классов, заключается в том, что вы можете слишком рано увязнуть в деталях реализации. Чтобы этому противостоять, концентрируйте внимание на концептуальной точке зрения и точке зрения спецификации. Если вы все же столкнулись с подобными проблемами, то в этом случае большую пользу могут оказать CRC-карточки (см. врезку в главе 5).

Где найти дополнительную информацию

Для более детального изучения диаграмм классов подходят все из упомянутых в главе 1 книг по общему описанию языка UML. Из более ранних книг мне особенно нравится книга Кука и Дэниелса, 1994 [13], поскольку авторы рассматривают в ней различные точки зрения и вводят для этого необходимый формализм.

5

Диаграммы взаимодействия

Диаграммы взаимодействия (interaction diagrams) представляют собой модели, предназначенные для описания поведения взаимодействующих групп объектов.

Как правило, диаграмма взаимодействия описывает поведение только одного варианта использования. На такой диаграмме отображаются только экземпляры объектов и сообщения, которыми они обмениваются между собой в рамках данного варианта использования.

Я проиллюстрирую данный подход на примере простого варианта использования, который обладает следующим поведением:

- Окно Ввода Заказа посылает Заказу сообщение «приготовиться».
- Заказ посылает данное сообщение каждой Строке Заказа в данном Заказе.
- Каждая Строка Заказа проверяет состояние определенного Запаса Товара.
  - Если данная проверка заканчивается успешно с результатом «истина» (true), то Строка Заказа удаляет соответствующее количество Запаса Товара и создает Позицию Доставки.
  - Если данная проверка заканчивается неудачей, т. е. количество Запаса Товара ниже требуемого уровня, то Запас Товара запрашивает новую поставку товара.

Существует два вида диаграмм взаимодействия: диаграммы последовательности (sequence diagrams) и диаграммы кооперации (collaboration diagrams).

Диаграммы последовательности

На **диаграмме последовательности** объекты изображаются прямоугольниками на вершине вертикальной пунктирной линии (рис. 5.1).

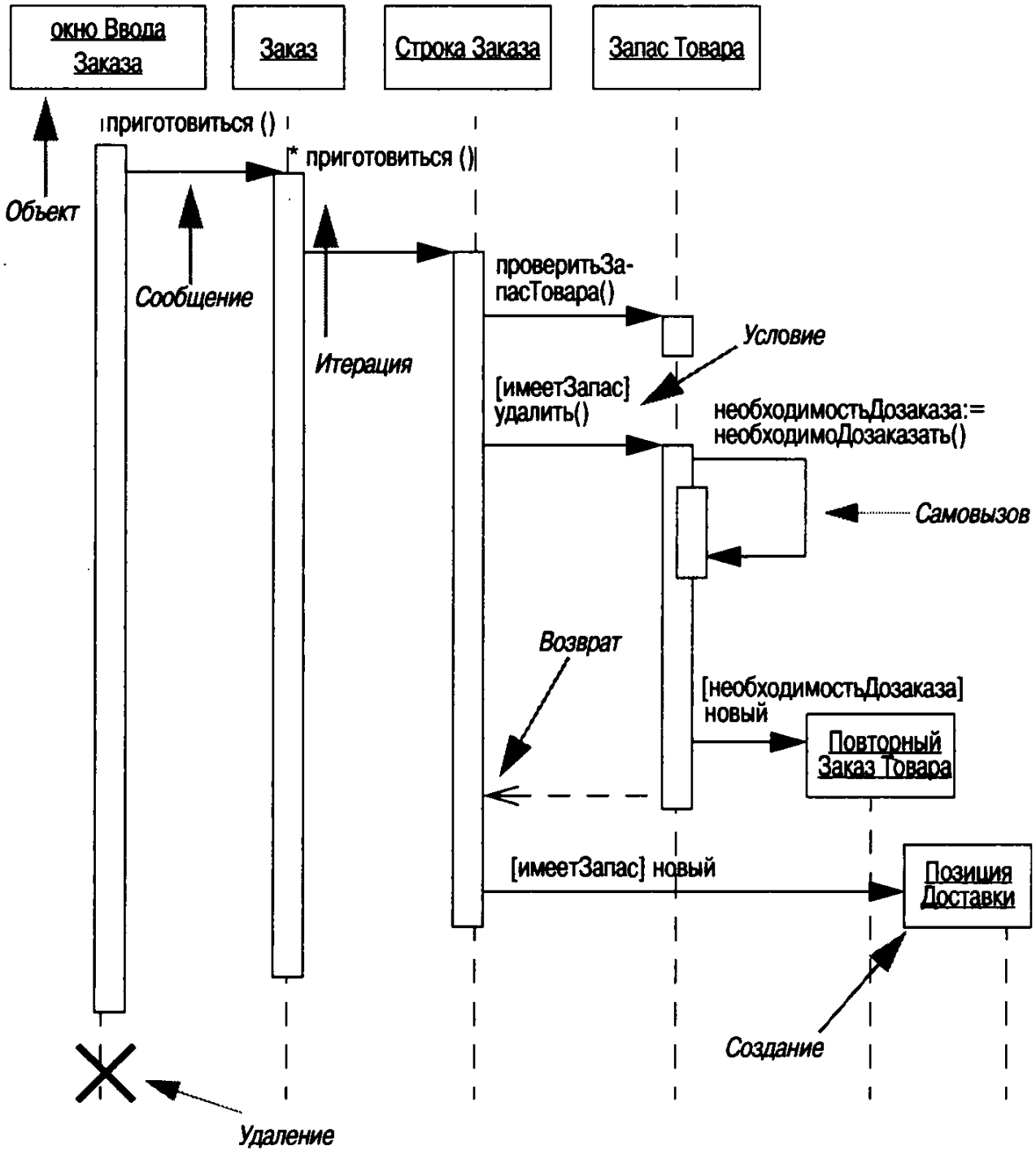


Рис. 5.1. Диаграмма последовательности

Эта вертикальная линия называется **линией жизни** (lifeline) объекта. Она представляет собой жизненный цикл объекта в процессе взаимодействия. Такая форма была впервые предложена А. Джекобсоном.

Каждое сообщение представляется стрелкой между линиями жизни двух объектов. Порядок следования сообщений устанавливается сверху вниз, то есть так, как они показываются на диаграмме. Каждое сообщение помечается как минимум именем сообщения; можно также

указать аргументы и некоторую управляющую информацию. На диаграмме последовательности могут присутствовать **рекурсивные вызовы** – сообщения, которые объекты посылают самим себе. При передаче такого сообщения стрелка указывает на ту же самую линию жизни.

Чтобы показать период времени, в течение которого объект является активным (для процедурного взаимодействия это следует указывать, когда процедура помещается в стек), изображается прямоугольник активности. Можно вообще не указывать прямоугольник активности; в этом случае диаграммы легче рисовать, но зато труднее понимать.

Управляющая информация может быть представлена двумя способами. Во-первых, существует некоторое **условие**, которое указывает, когда сообщение может быть передано (например, [*нуженПовторныйЗаказ*]). Сообщение посылается, только если это условие истинно. Условия могут оказаться полезными в простых случаях, однако в более сложных ситуациях я предпочитаю изображать для каждого случая отдельную диаграмму последовательности.

Во-вторых, может оказаться полезным некоторый управляющий маркер, называемый **маркером итерации**, показывающий, что сообщение посылается несколько раз для множества принимающих объектов. Такая итерация указывается в квадратных скобках с предшествующей звездочкой, например *[для всех позиций заказа].

На рис. 5.1 мы видим **возврат**, который указывает не новое сообщение, а возврат от переданного ранее сообщения. Возвраты отличаются от обычных сообщений тем, что они изображаются пунктирной линией. Некоторые разработчики изображают возврат для каждого сообщения, но, как мне кажется, это лишь загромождает диаграммы. Поэтому я их указываю только в том случае, когда уверен, что они приводят к большей ясности. Единственная причина, по которой изображен возврат на рис.5.1, – это демонстрация нотации. Если даже удалить возврат, диаграмма не станет менее понятной. Это хороший тест.

Как можно видеть, диаграмма последовательности на рис. 5.1 является очень понятной и наглядной. А это очень важно для разработчиков.

Одна из наиболее трудных проблем заключается в понимании организации потоков управления в объектно-ориентированной программе. Хороший проект может содержать много небольших методов в различных классах, и временами бывает довольно сложно понять последовательность поведения системы в целом. Можно долго вглядываться в код, пытаясь понять, что же делает программа. Особенно это характерно для тех, кто впервые сталкивается с объектным подходом. Диаграммы последовательности помогут вам разобраться в процессе поведения системы.

Диаграммы последовательности также имеют большое значение для моделирования параллельных процессов. На рис. 5.2 изображено несколько объектов, которые выполняют проверку банковской транзакции.

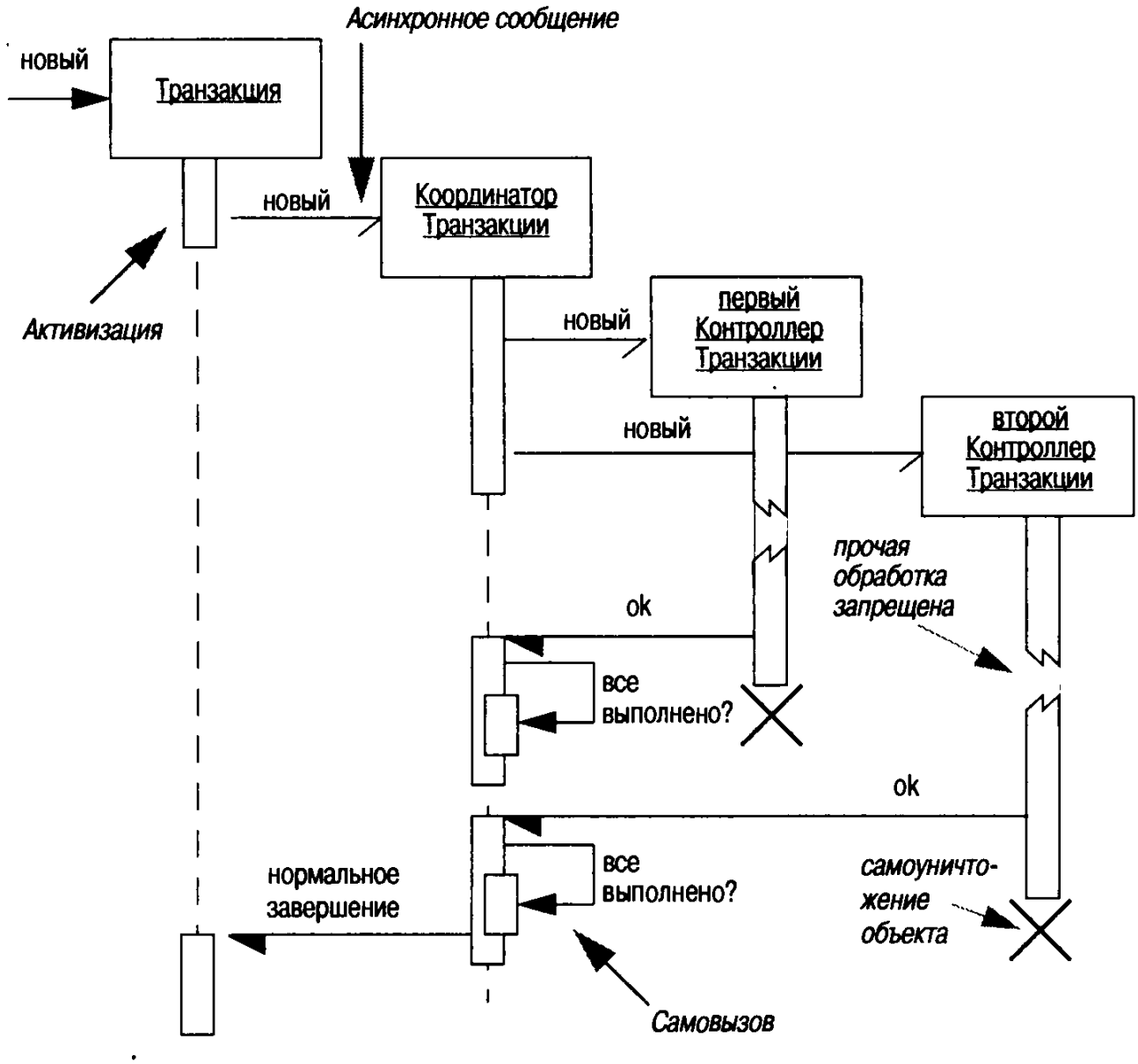


Рис. 5.2. Параллельные процессы и активизации

Когда создается некоторая Транзакция, она порождает Координатор Транзакции с целью координации проверки этой Транзакции. Этот координатор создает несколько (в данном случае два) объектов Контроллера Транзакции, каждый из которых несет ответственность за свою проверку. Этот процесс упрощает создание различных дополнительных процессов проверки, поскольку каждая проверка вызывается асинхронно и выполняется параллельно.

Когда Контроллер Транзакции завершает свою работу, он посылает сообщение Координатору Транзакции. Координатор проверяет, все ли Контроллеры сообщили о своих проверках. Если нет, то координатор не выполняет никаких действий. Если же все проверки завершились и завершились успешно, как в данном случае, то координатор посылает сообщение Транзакции о нормальном завершении.

Половина стрелки на конце сообщения служит для обозначения **асинхронного** сообщения. Асинхронное сообщение не блокирует вызывающий объект, то есть последний может продолжать выполнение

своего собственного процесса. Асинхронное сообщение может выполнять одно из трех действий:

1. Создание нового потока, при этом сообщение соединяется с прямоугольником активизации.
2. Создание нового объекта.
3. Установление связи с потоком, который уже выполняется.

Удаление объекта изображается большой буквой X. Объекты могут самоуничтожаться (как показано на рис. 5.2) либо могут быть уничтожены другим сообщением (рис. 5.3).

На рис. 5.2 и 5.3 показаны два сценария варианта использования «проверка транзакции». Каждый из этих сценариев изображен отдельно. Для включения условной логики в простую диаграмму существуют специальные методы. Но я предпочитаю не пользоваться ими, так как они слишком усложняют диаграмму.

На рис. 5.3 я применил очень полезный прием: поместил текстовые описания происходящих процессов вдоль левой части диаграммы по-

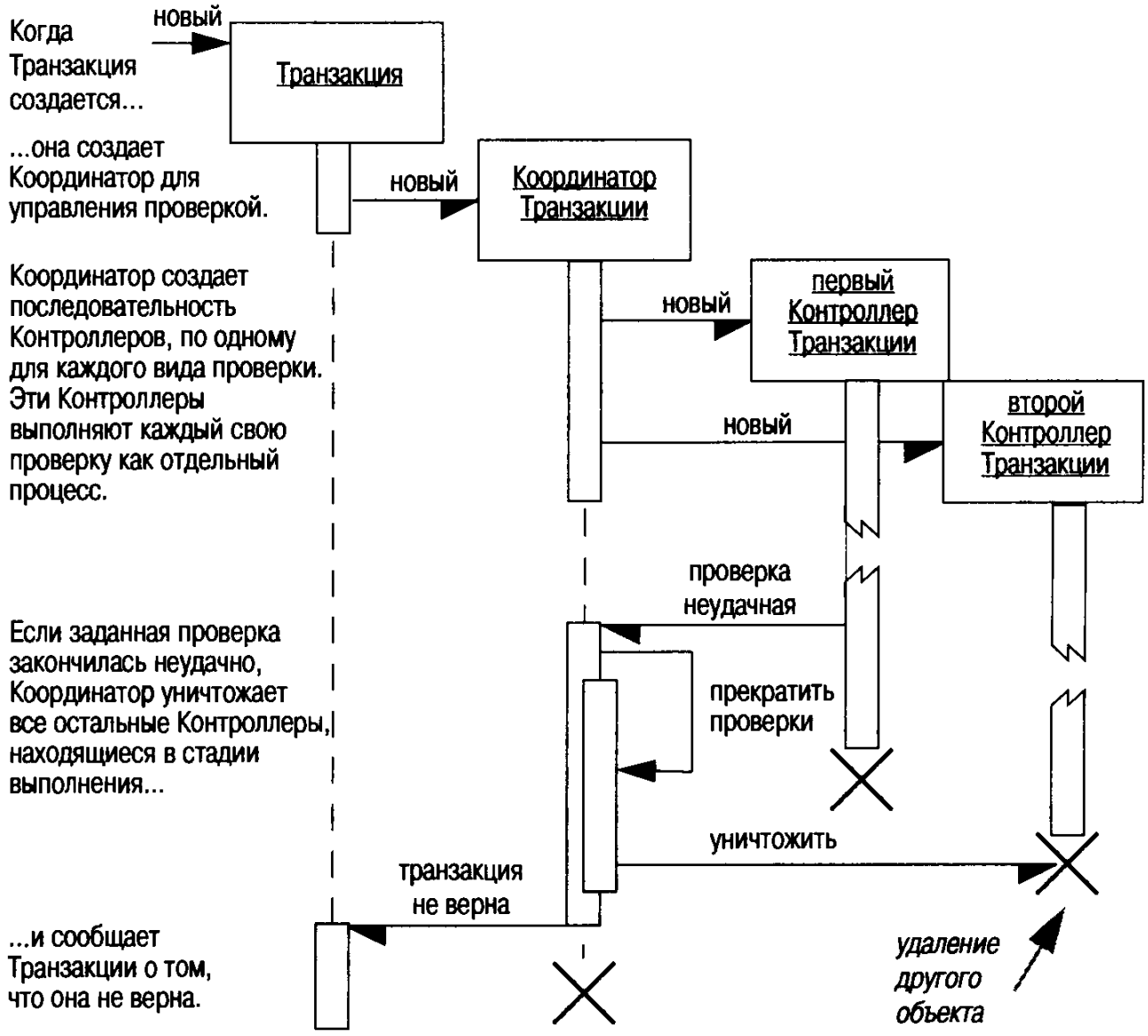


Рис. 5.3. Диаграмма последовательности: неудачная проверка

следовательности. При этом предполагается, что каждый блок текста располагается на одной горизонтальной линии с соответствующим сообщением на диаграмме. Это упрощает понимание диаграмм, хотя и требует некоторой дополнительной работы. Я использую этот прием для тех документов, которые собираюсь хранить.

Диаграммы кооперации

Еще одним видом диаграммы взаимодействия является **диаграмма кооперации.**

На такой диаграмме экземпляры объектов изображаются в виде пиктограмм. Так же как и на диаграмме последовательности, стрелки обозначают сообщения, обмен которыми осуществляется в рамках данного варианта использования. Однако их временная последовательность указывается посредством нумерации сообщений (рис. 5.4).

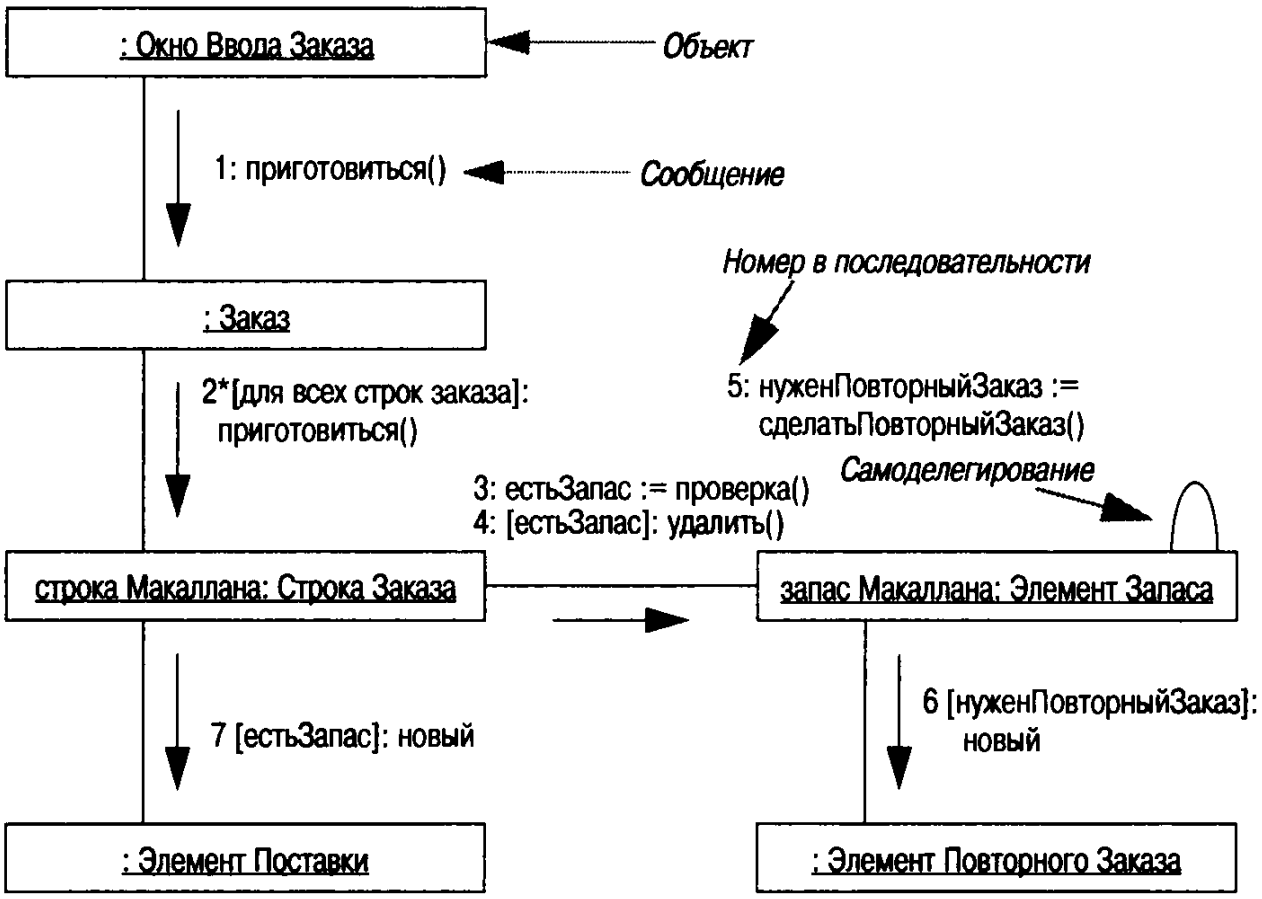


Рис. 5.4. Диаграмма кооперации с простой нумерацией

Подобная нумерация делает более трудным восприятие последовательности сообщений, чем в случае расположения линий на странице сверху вниз. С другой стороны, при таком пространственном расположении легче изобразить некоторые другие аспекты модели. Можно показать взаимосвязи объектов, схему использования перекрывающихся пакетов или другую информацию.

Для диаграмм кооперации можно использовать один из нескольких вариантов нумерации. Самый простой из них показан на рис. 5.4. Другой вариант десятичной нумерации представлен на рис. 5.5.

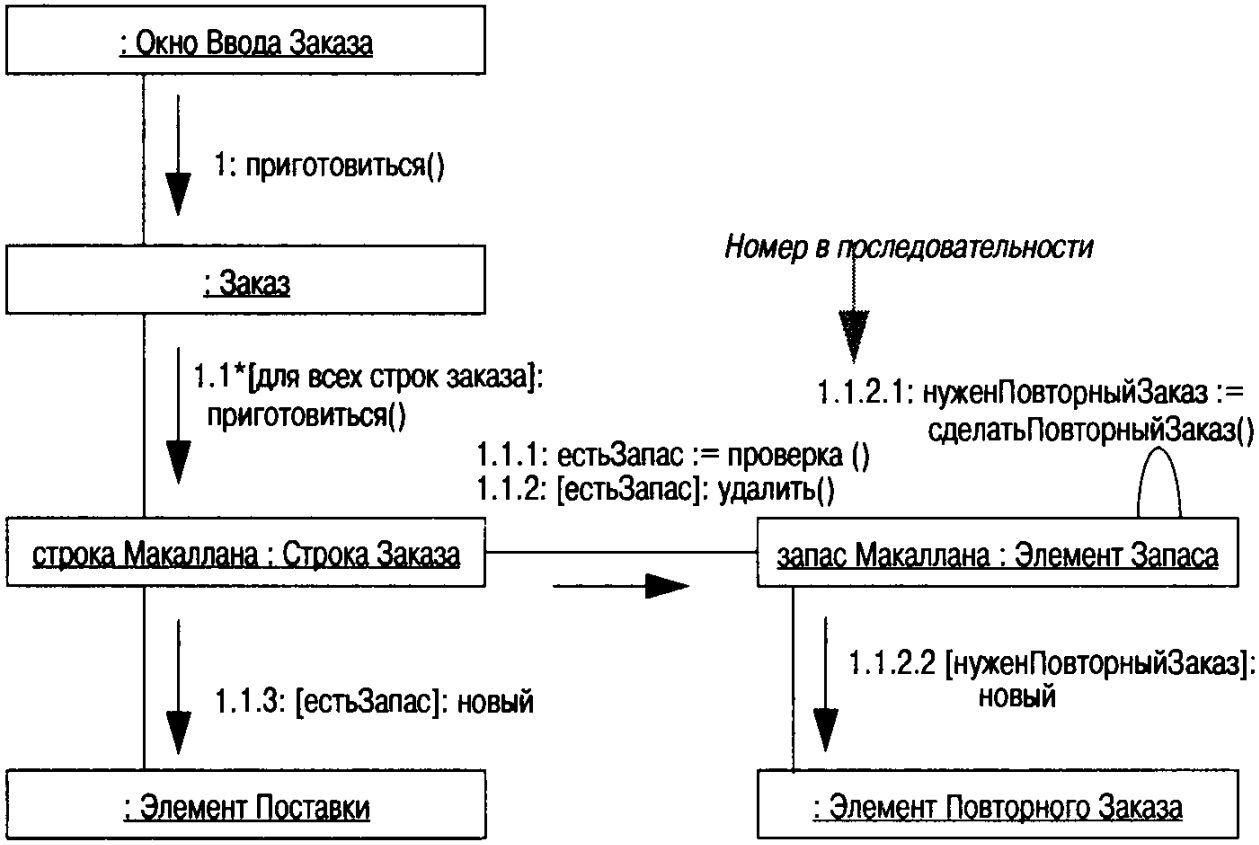


***Рис. 5.5.** Диаграмма кооперации с десятичной нумерацией*

Раньше разработчики пользовались простой схемой нумерации. В языке UML применяется десятичная схема нумерации, поскольку в этом случае понятно, какая из операций вызывает другую операцию, хотя при этом труднее разглядеть их последовательность в целом.

Независимо от используемой схемы нумерации на диаграмме кооперации можно разместить такого же рода управляющую информацию, как и на диаграмме последовательности.

На рис. 5.4 и 5.5 можно увидеть различные формы схем именования объектов в языке UML. Общая форма имеет вид *<ИмяОбъекта : ИмяКласса>*, где либо имя объекта, либо имя класса могут отсутствовать. При отсутствии имени объекта необходимо оставить двоеточие, чтобы было понятно, что это имя класса, а не объекта. Таким образом, имя «строка Макаллана: Строка Заказа» означает, что экземпляр класса Строка Заказа называется строка Макаллана (именно такой порядок записи имен мне особенно нравится). Я стараюсь именовать объекты в стиле языка Smalltalk, который я использовал в диаграммах последовательности. (Такая схема находится в соответствии с нотацией языка UML, поскольку «некоторыйОбъект» вполне подходит для имени некоторого объекта.)

Сравнение диаграмм последовательности и диаграмм кооперации

Разные разработчики имеют различные мнения по поводу выбора вида диаграммы взаимодействия. Я предпочитаю диаграмму последовательности, поскольку в ней сделан акцент именно на последовательности сообщений: легче наблюдать порядок, в котором происходят различные события. Другие предпочитают диаграммы кооперации, поскольку можно использовать схему пространственного расположения объектов для показа их статических взаимосвязей.

Одним из главных свойств любой формы диаграммы взаимодействия является их простота. Взглянув на диаграмму, сразу можно увидеть все сообщения. Однако если вы попытаетесь изобразить нечто более сложное, чем единственный последовательный процесс без множества условных переходов или циклов выполнения, то такая попытка закончится неудачей.

Одной из проблем при построении диаграмм взаимодействия являются возможные неудобства, связанные с представлением альтернативных ветвей процесса. Попытка изобразить альтернативы приводит к усложнению диаграмм и их многократной переработке. Поэтому для представления поведения я считаю очень полезными CRC-карточки.

CRC-карточки

В конце 80-х годов одним из крупнейших центров объектной технологии были исследовательские лаборатории фирмы Tektronix в Портленде, штат Орегон. В этих лабораториях была сосредоточена некоторая часть основных пользователей языка Smalltalk, и многие из главных идей объектной технологии родились именно там. Здесь же работали два таких известных программиста и специалиста по языку Smalltalk, как Уорд Каннингхем и Кент Бек. Они и сейчас занимаются методами обучения объектному языку Smalltalk. Результатом этой работы явился достаточно простой метод под названием «CRC-карточки» (Класс-Ответственность-Кооперация).

В то время как большинство аналитиков использовали для разработки моделей диаграммы, Уорд представлял описание классов на небольших карточках размером 4×6. Причем вместо атрибутов и методов класса он записывал на этих карточках ответственности (responsibilities).

Итак, что же такое ответственность? По существу, это высокоуровневое описание целевого назначения класса. Идея заключается в том, чтобы абстрагироваться от описания конкретных элементов данных и процессов и вместо этого несколькими предло-

жениями сформулировать целевое назначение класса. Небольшой размер карточки был выбран намеренно, чтобы описание было предельно кратким (рис. 5.6).

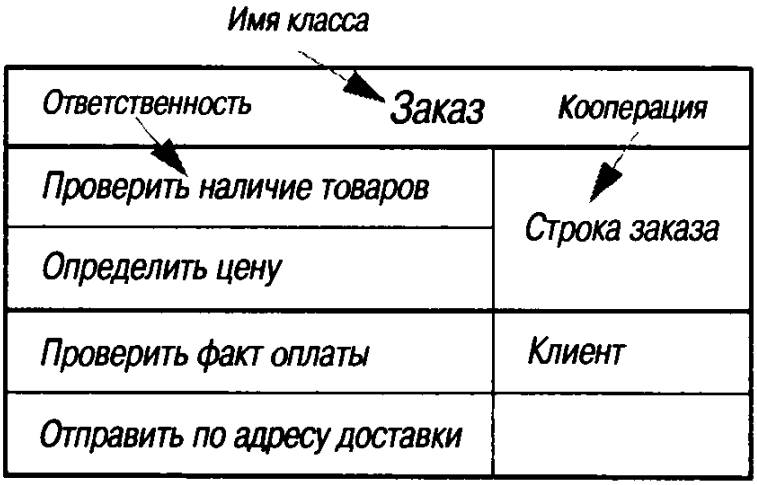


Рис. 5.6. Карточка Класс-Ответственность-Кооперация (CRC-card)

Вторая буква «С» соответствует кооперации классов. Для каждой ответственности вы показываете, с какими другими классами необходимо кооперироваться для ее реализации. Это дает вам некоторое представление о связях между классами, хотя и на достаточно высоком уровне.

Одно из главных достоинств CRC-карточек заключается в том, что они способствуют более оживленному обсуждению модели разработчиками. При работе над неким вариантом использования рассмотренные в этой главе диаграммы взаимодействия могут только замедлить процесс разработки, если нужно показать особенности реализации классов. Обычно вам необходимо рассмотреть альтернативы, а указанные диаграммы не позволяют наглядно их представить. С помощью CRC-карточек моделирование взаимодействия осуществляется посредством сортировки карточек по пачкам, а также их удаления. Это позволяет разработчикам быстро рассмотреть различные альтернативы.

По мере того как формируются представления относительно ответственности классов, их записывают на карточках. Концентрация внимания на ответственностях весьма важна, поскольку позволяет избавиться от рассмотрения классов как тупых хранителей данных и помогает команде разработчиков понять особенности высокоуровневого поведения каждого класса. Ответственность может соответствовать операции, атрибуту или (что более вероятно) неопределенному набору атрибутов и операций.

Наиболее распространенная ошибка разработчиков, с которой мне приходится сталкиваться, заключается в формировании слишком длинных списков низкоуровневых ответственностей.

Такая работа лишена смысла. Все ответственности должны легко помещаться на карточке. С моей точки зрения карточка, содержащая более трех ответственностей, вызывает дополнительные вопросы. В этом случае следует либо разделить класс на части, либо рассматривать ответственности на более высоком уровне.

Когда использовать CRC-карточки

Использование CRC-карточек помогает моделировать взаимодействие между классами, в частности, показать особенности выполнения некоторого сценария. Выяснив для себя детали взаимодействия, можно приступить к его документированию в форме диаграмм взаимодействия.

Использование ответственностей поможет вам сформулировать основные ответственности класса. А они являются полезным средством для приведения в порядок описаний классов.

Многие аналитики рекомендуют пользоваться методикой проигрывания ролей, когда каждый разработчик в команде играет роль одного или нескольких классов. Я никогда не видел, чтобы Уорд и Кент делали это, но сама идея этой методики мне нравится.

Где найти дополнительную информацию

Сэдли, Каннингхем и Бек никогда не писали книг о CRC, однако можно обратиться к их статье (Бек и Каннингхем, 1989) в Интернете по адресу: *http://c2.com/doc/oopsla89/paper.html*. Данный метод наилучшим образом описан в книге Вёфс-Брок (Wirfs-Brock), 1990 [46], в которой, по существу, изложена полная нотация использования ответственностей. Это довольно старая книга по объектно-ориентированному подходу, однако она хорошо себя зарекомендовала.

Когда следует использовать диаграммы взаимодействия

Диаграммами взаимодействия следует пользоваться в том случае, когда вы хотите описать поведение нескольких объектов в рамках одного варианта использования. Диаграммы взаимодействия удобны для изображения кооперации объектов и вовсе не так хороши для точного представления их поведения.

Если вы хотите описать поведение единственного объекта во многих вариантах использования, то следует применять диаграмму состояний (см. главу 8). Если же возникает необходимость описать поведение, охватывающее несколько вариантов использования или несколько нитей процесса, следует рассматривать диаграмму деятельности (см. главу 9).

6

Диаграммы классов: дополнительные понятия

Описанные ранее в главе 4 понятия соответствуют основной нотации диаграмм классов. Именно эти понятия нужно постичь и освоить прежде всего, поскольку они на 90% удовлетворят ваши потребности при построении диаграмм классов.

Однако диаграммы классов могут содержать множество нотаций для представления различных дополнительных понятий. Я сам использую их не слишком часто, но в отдельных случаях они оказываются весьма удобными. Рассмотрим последовательно эти дополнительные понятия, обращая внимание на особенности их применения. Однако при этом следует помнить, что их использование не является обязательным, и многим разработчикам удается вложить достаточно много смысла в диаграммы классов и без этих дополнительных понятий.

Возможно, при чтении этой главы вы столкнетесь с некоторыми трудностями. Могу вас обрадовать: вы можете без всякого ущерба пропустить эту главу при первом чтении книги и вернуться к ней позже.

Стереотипы

Стереотипы являются механизмом расширения ядра языка UML. Если для построения модели необходим некоторый элемент, и он отсутствует в языке UML, но похож на какую-либо конструкцию послед-

него, можно попытаться рассмотреть данный элемент в качестве стереотипа известной конструкции UML.

Примером подобной ситуации является интерфейс. В языке UML **интерфейс** представляет собой класс, который имеет только общедоступные операции без тел методов или атрибутов. Это соответствует интерфейсам в языках Java, COM и CORBA. Поскольку интерфейс является частным случаем класса, он определяется как стереотип класса. (Подробнее об этом см. в разделе «Интерфейсы и абстрактные классы» далее в этой главе.)

Стереотипы обычно записываются с помощью текста, заключенного в кавычки (например, «интерфейс»), однако они могут также изображаться с помощью пиктограммы стереотипа.

Многие расширения ядра языка UML можно описать в виде совокупности стереотипов. В рамках диаграмм классов могут существовать стереотипы классов, ассоциаций или обобщений. Вы можете понимать стереотипы как подтипы следующих типов метамодели: Класса, Ассоциации и Обобщения.

При использовании языка UML аналитикам иногда бывает трудно разобраться в разнице между ограничениями и стереотипами. Если вы помечаете класс как абстрактный, то что это – ограничение или стереотип? В существующих официальных документах это называется ограничением, но нужно сознавать, что различие между ним и стереотипом весьма размыто. Именно поэтому нет ничего удивительного в том, что подтипы зачастую являются более ограниченными, чем супертипы.

Работа OMG в этом направлении сосредоточена на создании так называемых профилей UML. **Профиль** представляет собой часть языка UML и расширяет его с помощью стереотипов, предназначенных для специальных целей. Начало этой работы OMG положено разработкой профиля реального времени и профиля языка определения интерфейсов (IDL) CORBA. Очевидно, что работа в этом направлении будет продолжена.

Диаграмма объектов

Диаграмма объектов представляет собой мгновенный снимок объектов системы с точки зрения времени. Поскольку на ней изображаются экземпляры классов, но не сами классы, диаграмма объектов часто называется диаграммой экземпляров.

Для представления варианта конфигурации объектов может быть использована некоторая диаграмма экземпляров. (На рис. 6.1 изображено некоторое множество классов, а на рис. 6.2 – соответствующее множество объектов.) Это может оказаться весьма полезным в случае сложных связей между объектами.

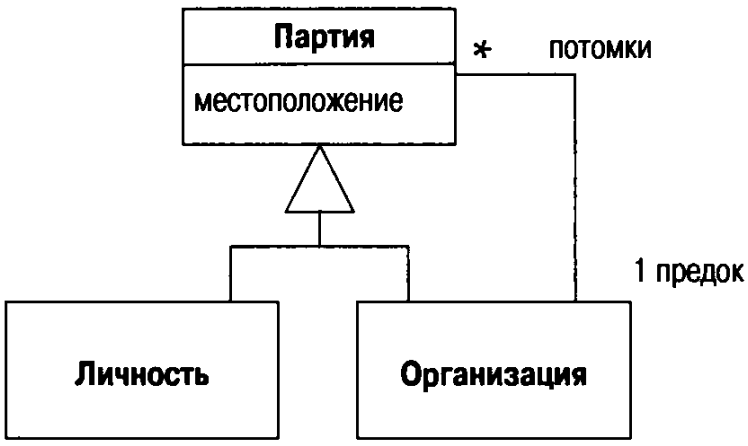


Рис. 6.1. Диаграмма классов для Партии сложной структуры

Вы можете сказать, что изображенные на рис. 6.2 элементы являются экземплярами, поскольку их имена подчеркнуты. Каждое имя записывается в виде *имя экземпляра : имя класса*. Обе части имени не являются обязательными, поэтому и Джон и :Личность представляют собой допустимые имена. При этом можно указать значения атрибутов и связей, как изображено на рис. 6.2.

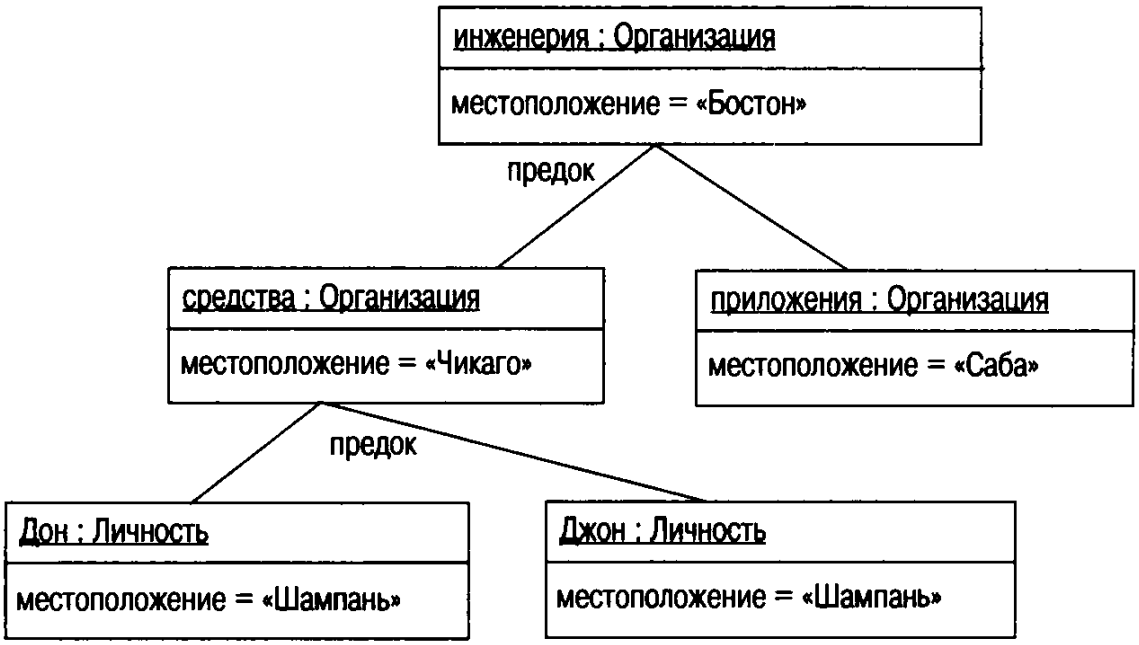


Рис. 6.2. Диаграмма объектов для примера экземпляров класса Партия

Диаграмму объектов можно представлять себе как диаграмму кооперации без сообщений.

Операции и атрибуты в контексте класса

Если в языке UML на операции или атрибуты ссылаются применительно к некоторому классу, а не экземпляру класса, то имеет место так называемый **контекст класса**. Это эквивалентно статическим членам в языках C++ или Java и переменным и методам класса в языке

Smalltalk. На диаграмме классов свойство в контексте класса подчеркивается (рис. 6.3).

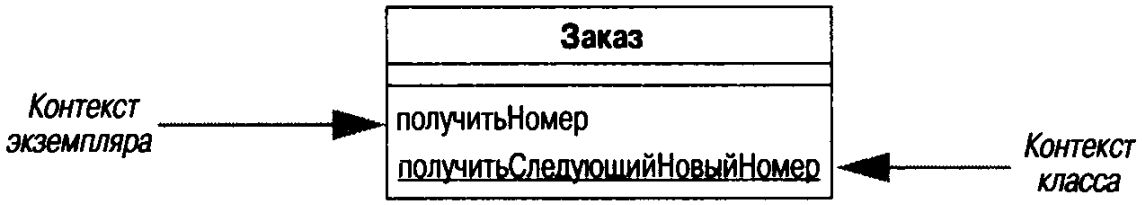


Рис. 6.3. Нотация контекста класса

Множественная и динамическая классификация

Классификация служит для обозначения отношения между некоторым объектом и его типом.

Большинство методов делают некоторые предположения о типе этого отношения; аналогичные предположения также присутствуют в основных объектно-ориентированных языках программирования. Джим Оделл высказал сомнения по поводу этих предположений, поскольку, по его мнению, они являются слишком жесткими для концептуального моделирования. Поскольку эти предположения устанавливают однозначную статическую классификацию объектов, Оделл предложил использовать для концептуального моделирования множественную динамическую классификацию.

При **однозначной классификации** любой объект принадлежит единственному типу, который может наследовать от супертипов. При **множественной классификации** объект может быть описан несколькими типами, которые не обязательно должны быть связаны наследованием.

Заметим, что множественная классификация отличается от множественного наследования. При множественном наследовании тип может иметь много супертипов, но для каждого объекта должен быть определен только один тип. Множественная классификация допускает принадлежность объекта нескольким типам без определения специального типа для этой цели.

В качестве примера рассмотрим тип Личность, подтипами которой являются Мужчина или Женщина, Доктор или Медсестра, Пациент или вообще никто (рис. 6.4). Множественная классификация позволяет некоторому объекту иметь любой из этих типов в любом допустимом сочетании, при этом нет необходимости определять типы для всех допустимых сочетаний.

Если вы используете множественную классификацию, то должны быть уверены в том, что четко определили, какие сочетания являются допустимыми. Это делается с помощью **дискриминатора**, который помечает линию обобщения и характеризует сущность подтипов. Не-

сколько подтипов могут иметь общий для всех дискриминатор. Все подтипы с одним и тем же дискриминатором являются непересекающимися, то есть любой экземпляр супертипа может быть экземпляром только одного из подтипов с данным дискриминатором. Удачный способ изобразить ситуацию, в которой несколько подклассов используют один дискриминатор, – это соединить соответствующие линии с одним треугольником обобщения, как показано на рис. 6.4. Альтернативный способ – изобразить несколько стрелок с одинаковыми текстовыми метками.

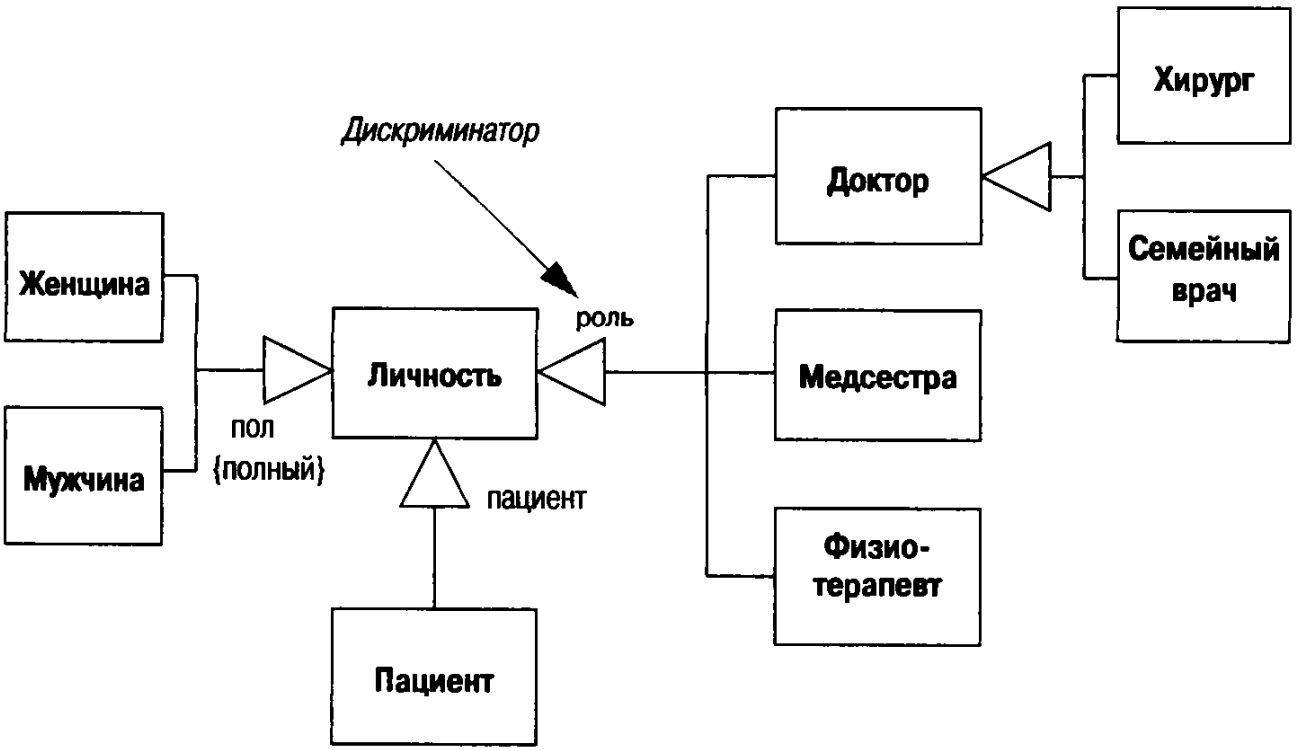


Рис. 6.4. Множественная классификация

Можно задать одно полезное ограничение, которое позволяет утверждать, что любой экземпляр суперкласса должен быть экземпляром одного из подклассов в данной группе. (Суперкласс в этом случае является абстрактным.) Хотя по этому вопросу в стандарте языка UML существует некоторая путаница, многие аналитики используют ограничение {complete} (полный) для задания подобного ограничения.

В качестве иллюстрации отметим следующие допустимые сочетания подтипов на диаграмме: (Женщина, Пациент, Медсестра), (Мужчина, Физиотерапевт), (Женщина, Пациент) и (Женщина, Доктор, Хирург). Заметим также, что такие сочетания, как (Пациент, Доктор) и (Мужчина, Доктор, Медсестра) являются недопустимыми. Первое множество недопустимо, поскольку не включает тип, определенный дискриминатором «пол» с ограничением {полный}; второе множество недопустимо, поскольку включает сразу два типа с одним и тем же дискриминатором «роль». По определению однозначной классификации соответствует единственный непомеченный дискриминатор.

Возникает еще один вопрос: может ли объект изменять свой тип? Примером такой ситуации является банковский счет. Когда счет клиента становится пустым, он существенно меняет свое поведение. В частности, должны быть переопределены некоторые операции (включая «снять со счета» и «закрыть счет»).

Динамическая классификация разрешает объектам изменять свой тип в рамках структуры подтипов, а **статическая классификация** этого не допускает. Статическая классификация проводит границу между типами и состояниями, а динамическая классификация объединяет эти понятия.

Следует ли использовать множественную динамическую классификацию? Я полагаю, что она полезна для концептуального моделирования. Ее можно использовать и при моделировании спецификаций, но при этом нужно иметь подходящее средство для ее реализации. Весьма характерно, что с точки зрения интерфейса динамическая классификация выглядит как обычное обобщение, и пользователь класса не сможет определить, какая конкретная реализация используется (см. Фаулер, 1997 [18]). Однако, как и в большинстве подобных случаев, выбор зависит от конкретных обстоятельств, и окончательное решение остается за вами. Преобразование множественного динамического интерфейса в однозначную статическую реализацию может оказаться более затруднительным, чем оно того заслуживает.

На рис. 6.5 показан пример использования динамической классификации в отношении работы, выполняемой личностью, которая, естественно, может меняться. Характерно, что в этом случае для подтипов необходимо определить дополнительное поведение, а не только одни метки. В подобных случаях зачастую имеет смысл создать отдельный класс для выполняемой работы и связать этот класс с личностью с помощью некоторой ассоциации. Специально для этой цели я разработал образец под названием «Ролевые Модели»; информацию об этом образце и другую информацию, дополняющую мою книгу «Analysis Patterns» [18], можно найти в Интернете на моей домашней страничке.

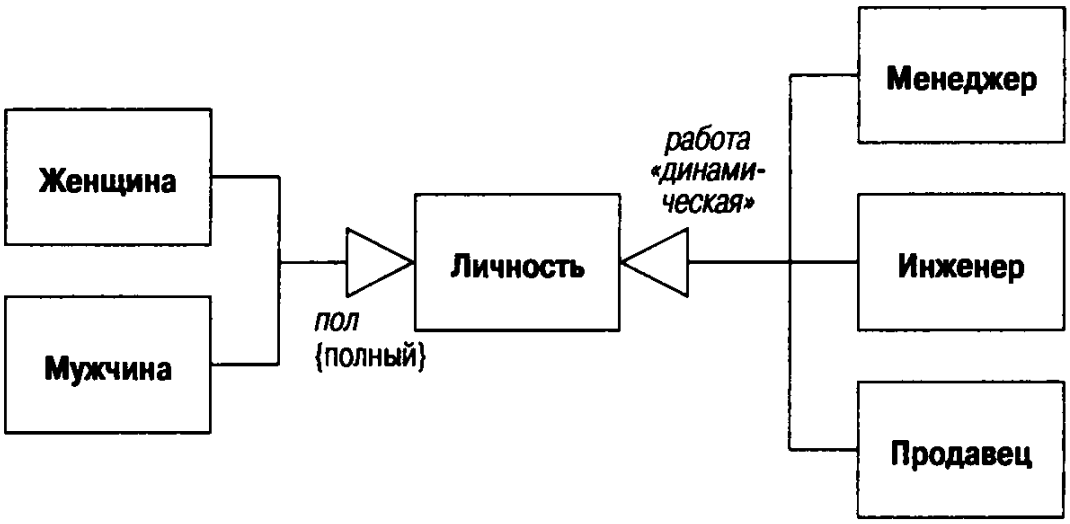


Рис. 6.5. Динамическая классификация

Агрегация и композиция

Агрегация – один из моих излюбленных приемов моделирования. Это легко объяснить в двух словах: **агрегация** – отношение «являться частью». Например, можно сказать, что двигатель и колеса являются частями автомобиля. Звучит вроде бы просто, однако при рассмотрении разницы между агрегацией и ассоциацией возникают определенные трудности.

До появления языка UML вопрос о различии агрегации и ассоциации у аналитиков просто не возникал. Осознавалась подобная неопределенность или нет, но свои работы в этом вопросе аналитики совсем не согласовывали между собой. В результате многие разработчики считали агрегацию важной, но по совершенно другой причине. Язык UML определяет агрегацию, но семантика этого отношения очень широка. Вот мнение Джима Рамбо: «Представляйте ее как безвредное лекарство» (Рамбо, Джекобсон, Буч, 1999, [37]).

В дополнение к агрегации в языке UML определена более сильная разновидность агрегации, называемая композицией. При **композиции** объект-часть может принадлежать только единственному целому; кроме того, как правило, жизненный цикл частей совпадает с жизненным циклом целого: части живут и умирают вместе с целым. Обычно любое удаление целого распространяется на все его части.

Такое каскадное удаление часто рассматривается как часть определения агрегации, однако оно имеет место только в том случае, когда кратность конца ассоциации составляет 1..1. Например, если вы действительно хотите удалить Клиента, то должны распространить это удаление на Заказы (и, соответственно, на Строки Заказа).

На рис. 6.6 изображены примеры рассмотренных ранее понятий. Связи композиции у Точки означают, что любой экземпляр *Точки* может быть либо *Многоугольником*, либо *Окружностью*, но не может быть

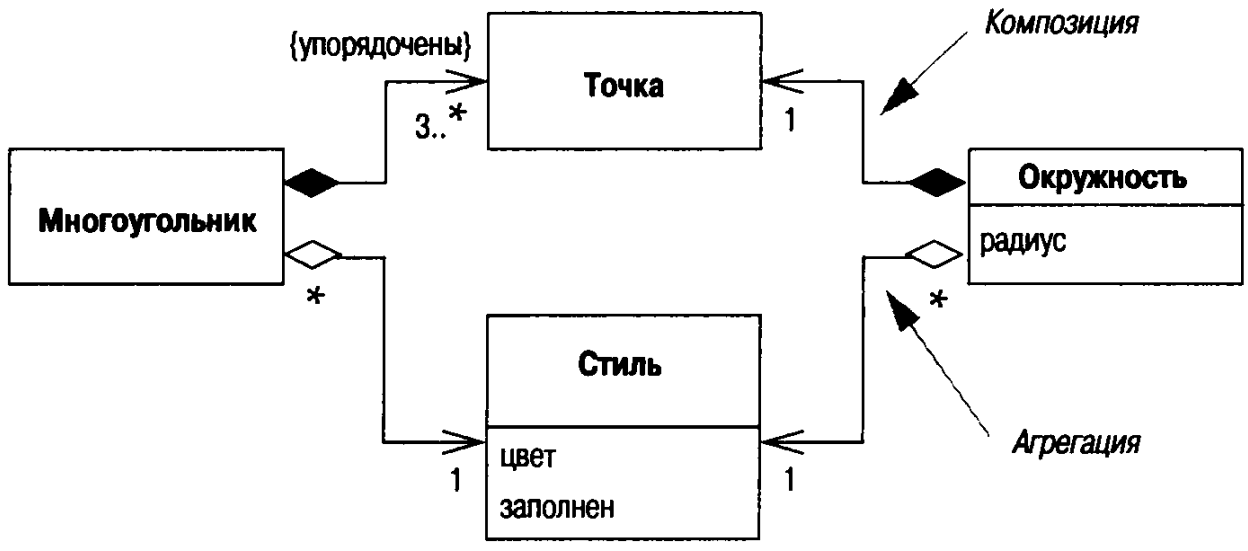


Рис. 6.6. Агрегация и композиция

ими одновременно. Однако некоторый экземпляр *Стиля* может быть общим для нескольких *Многоугольников* и *Окружностей*. Более того, указанная композиция означает, что удаление некоторого *Многоугольника* повлечет за собой удаление всех ассоциированных с ним *Точек*, но *не* повлечет удаление связанного с ним *Стиля*.

Подобное ограничение, а именно: некоторая *Точка* может одновременно принадлежать только либо *Многоугольнику*, либо *Окружности*, не может быть представлено с помощью кратности 1. Из этого также следует, что точка является объектом-значением (см. раздел «Ссылочные объекты и объекты-значения» в этой главе). К противоположной стороне этой ассоциации можно добавить кратность 0..1, но сам я этого не делаю. Черный ромб говорит все, что должно быть сказано.

На рис. 6.7 показано другое обозначение композиции. В этом случае компонент-часть помещается внутрь целого. Имя классса-компонента не выделяется полужирным шрифтом и записывается в виде *имя роли*: *имя Класса*. Кроме того, в верхнем правом углу указывается кратность. Можно использовать также и атрибут для компоненты с единственным значением.

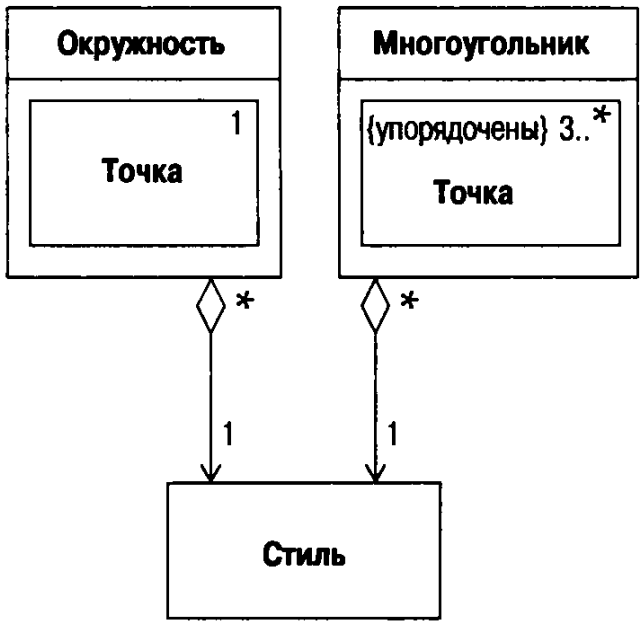


Рис. 6.7. Альтернативная нотация для композиции

В разных ситуациях для композиции используются различные обозначения. Это только две нотации для композиции, хотя вариантов для подобной нотации, предлагаемых языком UML, гораздо больше. Следует обратить внимание, что эти варианты нотации могут использоваться только для композиции, но не для агрегации.

Производные ассоциации и атрибуты

Производные ассоциации и **производные атрибуты** могут быть получены, соответственно, из других ассоциаций и атрибутов на диаграм-

ме классов. Например, значение атрибута возраст для Личности можно определить, если известна дата рождения этой Личности.

Каждая точка зрения предполагает свою собственную интерпретацию производных ассоциаций и атрибутов на диаграмме классов. Наиболее важная особенность связана с точкой зрения спецификации. С этой точки зрения важно понимать, что производные ассоциации и атрибуты устанавливают лишь ограничения между значениями, а не определяют то, что вычисляется и хранится постоянно.

На рис. 6.8 изображена некоторая иерархическая структура счетов с позиции спецификации. В данной модели используется образец *Композиция* (см. Гамма и др., 1995).

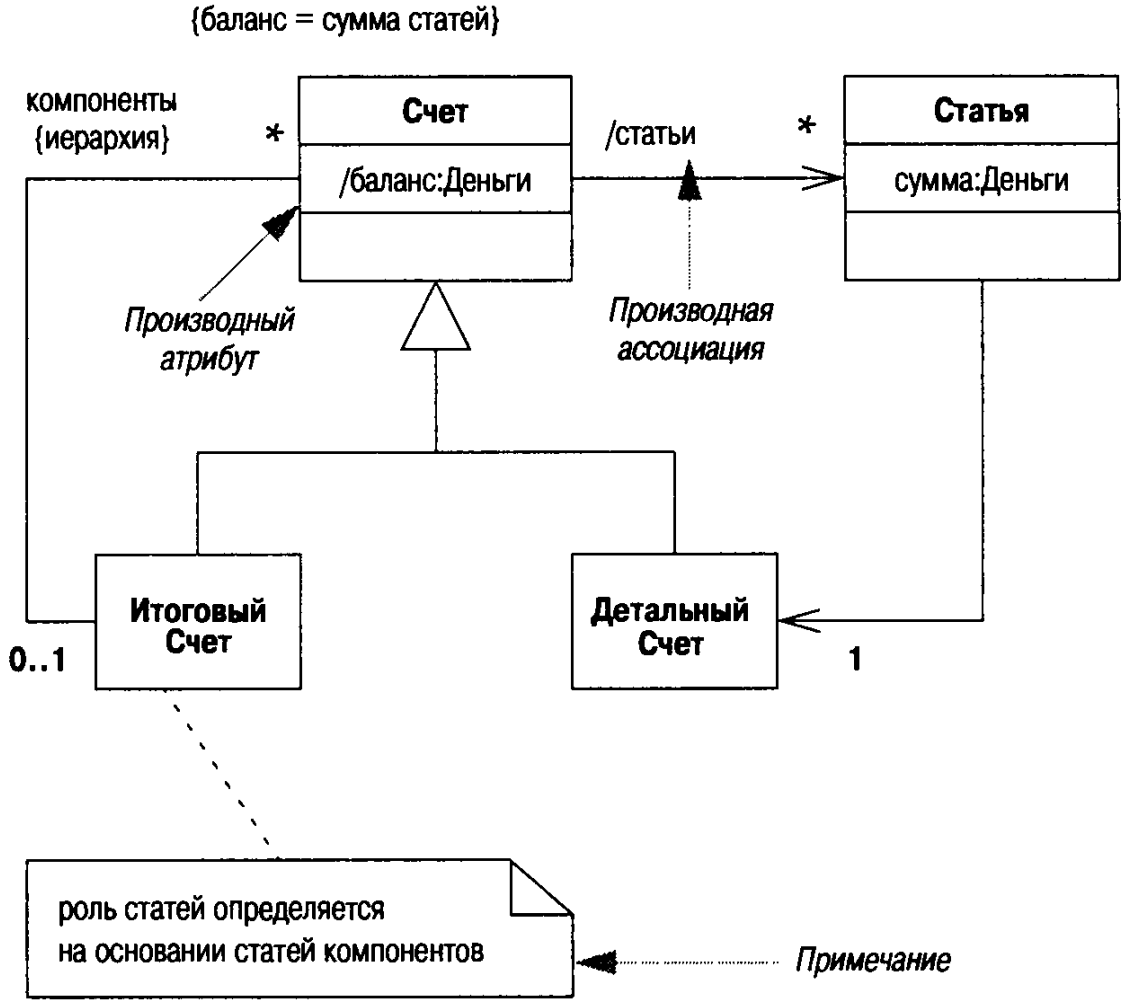


Рис. 6.8. Производные ассоциации и атрибуты

Отметим следующее:

- Объекты класса Статья связаны с Детальными Счетами.
- Баланс некоторого Счета вычисляется как сумма его Статей.
- Статьи Итогового Счета являются статьями его компонентов, которые определяются рекурсивно.

Поскольку на рис. 6.8 изображена модель уровня спецификации, она ничего не утверждает относительно того, содержат ли Счета поля для

хранения данных баланса. Такие данные могут присутствовать в модели, однако они скрыты от клиентов класса Счет.

На рис. 6.9 показан пример производных элементов, которые задают ограничения для класса с именем Период Времени.

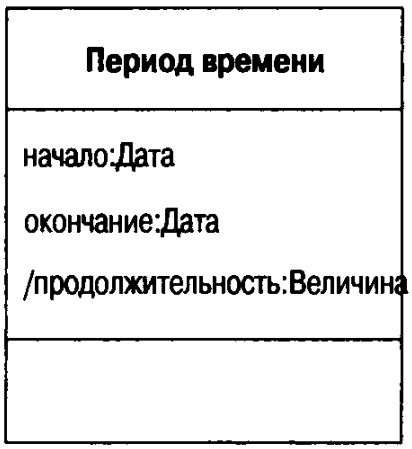


Рис. 6.9. Класс Период Времени

Если это диаграмма уровня спецификации, то, хотя и предполагается, что *начало* и *окончание* являются хранимыми атрибутами, а *продолжительность* – вычисляемым атрибутом, реально программист может реализовать этот класс любым способом, лишь бы при этом обеспечивалось его внешнее поведение. Например, вполне возможно сделать хранимыми атрибуты *начало* и *продолжительность*, а атрибут *окончание* – вычисляемым.

На диаграмме уровня реализации производные значения являются важными для специально выделенных полей, используемых в качестве кэш-памяти для повышения производительности. Помечая эти поля и записывая производные значения в кэш, можно легко видеть, как используется кэш-память. Как правило, я закрепляю назначение таких полей с помощью слова «кэш» в их именах в коде (например, *балансКэш*).

На концептуальных диаграммах я использую производные маркеры, для того чтобы они напоминали мне, где эти производные элементы существуют и что нужно согласовать этот факт с экспертами предметной области. Затем все должно быть согласовано с использованием этих элементов на соответствующих диаграммах уровня спецификации.

Интерфейсы и абстрактные классы

Одно из наиболее важных свойств объектно-ориентированной разработки заключается в возможности изменения интерфейсов классов независимо от их реализации. Именно благодаря этому свойству объектная разработка является столь мощным средством. Однако очень немногие разработчики используют это свойство должным образом.

В языках программирования используется единственная объектная конструкция – класс, который включает как интерфейс, так и реализацию. Если вы определяете подкласс, то он наследует и то, и другое. К сожалению, разработчики крайне редко используют интерфейс как отдельную конструкцию.

Интерфейс в чистом виде, как, например, в языке Java, представляет собой класс без каких-либо деталей реализации и поэтому содержит только определения операций, но не тела методов и не поля. Часто интерфейсы определяются на основе абстрактных классов. Такие классы могут предполагать некоторую реализацию, но в большинстве случаев они используются в основном для определения интерфейса. Дело заключается в том, что механизм подклассов или какой-либо другой механизм может обеспечить реализацию, однако клиенты никогда не должны видеть ее, а только интерфейс.

Типичным примером подобной ситуации является текстовый редактор, изображенный на рис. 6.10. Чтобы сделать редактор независимым от платформы, мы определяем независимый от платформы абстрактный класс Окно. Этот класс не содержит тел методов; он определяет только интерфейс для использования в текстовом редакторе. За-

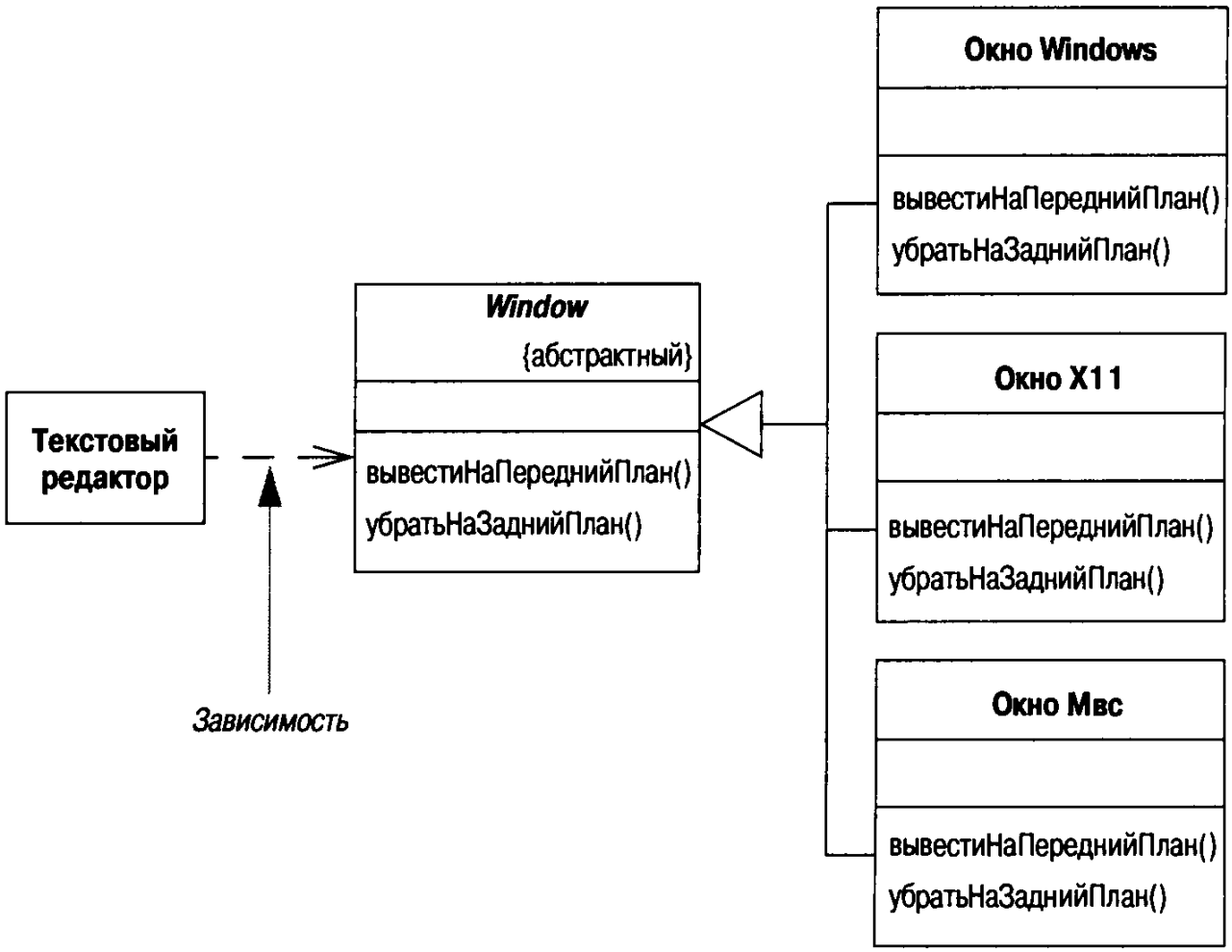


Рис. 6.10. Окно как абстрактный класс

висимые от платформы подклассы могут использоваться как производные элементы.

Если вы определяете абстрактный класс или метод, то язык UML требует выделять его имя курсивом. Можно также (или вместо курсива) использовать ограничение {abstract} (абстрактный). На бумаге я обычно использую это ограничение, поскольку не умею писать курсивом. Однако я предпочитаю изящество курсива, если пользуюсь инструментальным средством для построения диаграмм.

Тем не менее, механизм подклассов отнюдь не является единственно возможным средством определения интерфейса. В языке Java интерфейс определяется как самостоятельная конструкция, и компилятор проверяет, чтобы в секции реализации класса реализовывались все операции, определенные в секции интерфейса данного класса.

На рис. 6.11 показаны классы ВходнойПоток, ВводДанных и ВходнойПотокДанных (определяемые в стандартном пакете *java.io*). При этом ВходнойПоток является абстрактным классом, а ВводДанных – интерфейсом.

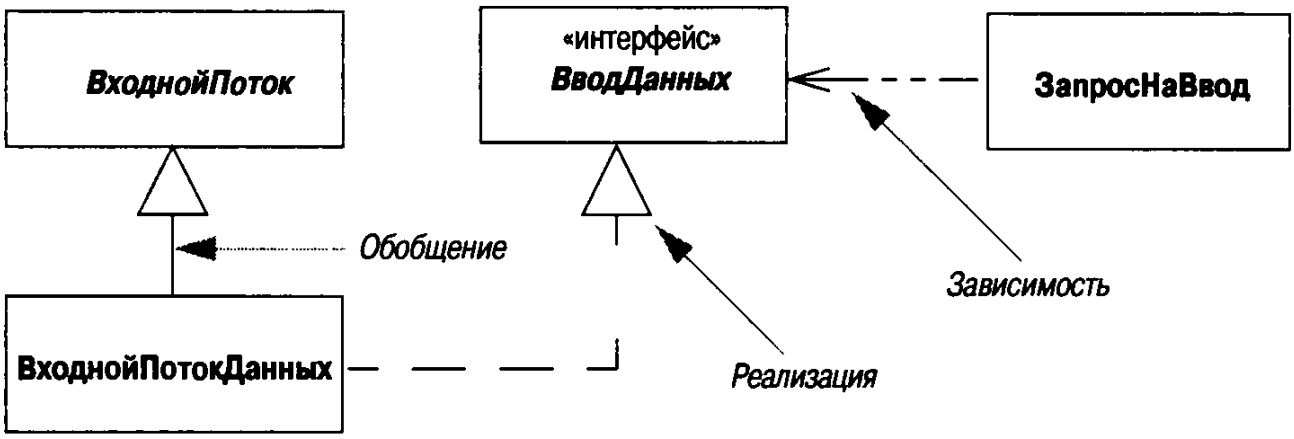


Рис. 6.11. Интерфейсы и абстрактный класс: пример на языке Java

Предположим, некоторому классу-клиенту, скажем ЗапросНаВвод, требуется воспользоваться функциональностью класса ВводДанных. Класс ВходнойПотокДанных реализует интерфейсы как класса ВводДанных, так и класса ВходнойПоток, являясь при этом подклассом последнего.

Связь между классами ВходнойПотокДанных и ВводДанных представляет собой отношение **реализации**. Реализация сознательно изображается аналогично обобщению; она указывает, что один из классов реализует поведение, специфицированное в другом классе. Допускается ситуация, когда секция реализации одного класса реализует другой класс; это означает, что реализующий класс должен согласовывать свой интерфейс, но вовсе не должен использовать наследование.

В модели уровня спецификации отсутствует различие между реализацией и механизмом подтипов.

Связь между классами ЗапросНаВвод и ВводДанных представляет собой отношение **зависимости** (dependency). В данной ситуации она показывает, что если изменится интерфейс класса ВводДанных, то класс ЗапросНаВвод также может измениться. Одна из целей разработки состоит в сведении к минимуму количества зависимостей, для того чтобы воздействие подобных изменений было минимальным. (Более подробно о зависимостях будет сказано в главе 7.)

Альтернативная и более компактная нотация изображена на рис. 6.12. Здесь интерфейсы представлены в форме маленьких кружков (называемых часто леденцами на палочках), которые соединены с реализующими их классами.

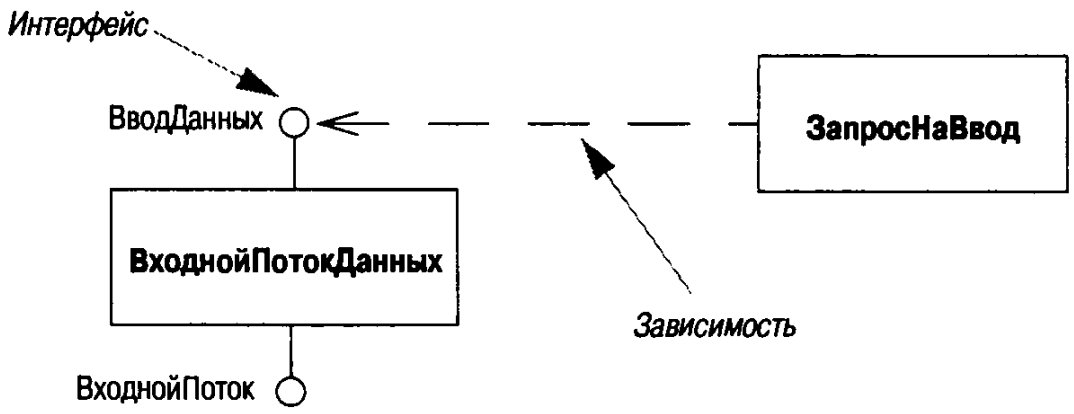


Рис. 6.12. Графическая нотация для интерфейсов

При использовании обозначения, похожего на леденцы на палочках, не существует различия между реализацией интерфейса и механизмом подклассов для некоторого абстрактного класса. Хотя эта нотация и является более компактной, в этом случае нельзя увидеть ни операций интерфейса, ни каких-либо отношений обобщения между интерфейсами.

Абстрактные классы и интерфейсы похожи, однако между ними существует различие. Как те, так и другие позволяют определить интерфейс и отложить его реализацию на более позднее время. Однако абстрактный класс позволяет дополнить реализацию некоторых методов, в то время как интерфейс вынуждает отложить определение всех методов.

Ссылочные объекты и объекты-значения

Одно из наиболее общих свойств объектов заключается в том, что они обладают индивидуальностью (identity). Это так, но все обстоит не столь просто, как может показаться. На практике индивидуальность важна для ссылочных объектов и не столь важна для объектов-значений.

Ссылочные объекты (reference objects) – это такие объекты, как Клиент. В данном случае индивидуальность очень важна, поскольку в ре-

альном мире конкретному клиенту обычно должен соответствовать только один программный объект. Любой объект, обратившийся к объекту Клиент, может воспользоваться соответствующей ссылкой или указателем; в результате все объекты, обратившиеся к данному Клиенту, получат доступ к одному и тому же программному объекту. Аналогично изменения, вносимые в объект Клиент, становятся доступными всем пользователям этого объекта.

Если у вас имеются две ссылки на объект Клиент и вы хотите установить их тождественность, то обычно сравниваете индивидуальность тех объектов, на которые указывают эти ссылки. Создание копий объектов может быть запрещено, но даже если оно разрешено, к нему стараются прибегать как можно реже, возможно, только с целью архивирования или репликации в сети. Если все же копии созданы, то необходимо обеспечить синхронизацию вносимых в них изменений.

Объекты-значения (value objects) – это такие объекты, как Дата. Как правило, один и тот же объект в реальном мире может быть представлен целым множеством объектов-значений. Например, вполне нормально, когда имеются сотни объектов со значением «1 января 1999 года». Все эти объекты являются взаимозаменяемыми копиями. При этом новые даты создаются и уничтожаются достаточно часто.

Если у вас имеются две даты и вы хотите установить, являются ли они тождественными, то вполне достаточно просто посмотреть на их значения, а не устанавливать их индивидуальность. Обычно это означает, что в программе необходимо определить оператор проверки равенства, который бы проверял для дат год, месяц и день (каким бы ни было их внутреннее представление). Обычно каждый объект, который ссылается на 1 января 1999 года, имеет свой собственный специальный объект, однако иногда даты могут быть объектами общего пользования.

Объекты-значения должны быть постоянными (неизменяемыми; см. раздел «Постоянство» далее в этой главе). Другими словами, не должно допускаться изменение значения объекта-даты «1 января 1999 года» на «2 января 1999 года». Вместо этого следует создать новый объект «2 января 1999 года» и связать его с первым объектом. Причина запрета подобного изменения заключается в следующем: если бы эта дата была объектом общего пользования, то ее обновление могло повлиять на другие объекты-даты непредсказуемым образом.

В прежнее время различие между ссылочными объектами и объектами-значениями было более четким. Объекты-значения являлись встроенными значениями системы типов. В настоящее время системы типов можно расширять с помощью своих собственных классов, поэтому данный аспект требует более внимательного отношения. В рамках языка UML для объектов-значений обычно используются атрибуты, а для ссылочных объектов – ассоциации. Для объектов-значений можно также использовать композицию.

Я не думаю, что различие между ссылочными объектами и объектами-значениями следует учитывать при концептуальном моделировании. Это может привести к путанице с кратностями. Если я представляю связь с объектом-значением с помощью некоторой ассоциации, то обычно обозначаю кратность на конце такой ассоциации около пользователя данного значения символом «*». Это происходит лишь в том случае, если для него не существует правила уникальности, такого как использование порядкового номера.

Совокупности многозначных концов ассоциаций

Многозначный конец ассоциации – это такой конец ассоциации, верхняя граница кратности которого превышает 1 (например, «*»). Обычно принято рассматривать такие многозначные концы ассоциаций как обычные множества. При этом для соответствующих целевых объектов не существует никакой упорядоченности, и не существует объектов, которые могли бы присутствовать в данном множестве более одного раза. Однако эти правила могут быть изменены с помощью введения дополнительного ограничения.

Ограничение {*упорядочено*} ({*ordered*}) устанавливает некоторый порядок на множестве целевых объектов, т. е. целевые объекты образуют список. В этом списке каждый целевой объект может присутствовать только один раз.

Я использую ограничение {*комплект*} ({*bag*}), чтобы показать, что целевые объекты на данном конце ассоциации могут появляться более одного раза, но без какой-либо упорядоченности. Я также использую ограничение {*иерархия*} ({*hierarchy*}), чтобы показать наличие некоторой иерархии в множестве целевых объектов, и ограничение {*dag*} (сокращение от directed acyclic graph), характеризующее направленный ациклический граф.

Постоянство

Постоянство (frozen) представляет собой ограничение, которое в языке UML может быть применено по отношению к атрибуту или концу ассоциации, но, по моему мнению, оно также оказывается полезным применительно к классам.

По отношению к атрибуту или концу ассоциации постоянство указывает на то, что значение этого атрибута или конца ассоциации не может быть изменено в течение всего жизненного цикла исходного объекта. Это значение должно быть задано при создании объекта и после этого уже никогда не может изменяться. Начальное значение может быть неопределенным (null). Разумеется, если данное ограниче-

ние справедливо при конструировании объекта, оно будет оставаться справедливым в течение всего времени существования объекта. При этом подразумевается, что для данного значения должен присутствовать аргумент в конструкторе и не должно существовать операций, которые могли бы изменить это значение.

По отношению к классу постоянство указывает, что все концы ассоциации и атрибуты, связанные с данным классом, являются постоянными.

Постоянство и ограничение «только для чтения» (read-only) – это не одно и то же. «Только для чтения» предполагает, что соответствующее значение нельзя изменить непосредственно, однако оно может быть изменено вследствие изменения какого-либо другого значения. Например, если атрибутами личности являются дата рождения и возраст, то возраст может иметь ограничение «только для чтения», но не может быть постоянным.

Я обозначаю постоянство с помощью ограничения {*постоянно*} ({*frozen*}) и помечаю значения, предназначенные только для чтения, с помощью ограничения {*только для чтения*}. (При этом следует заметить, что Только Для Чтения не является стандартным элементом языка UML.)

Если вы собираетесь определить что-либо как постоянное, следует помнить, что люди способны совершать ошибки. При разработке программного обеспечения человек моделирует только то, что сам знает о реальном мире, не учитывая всей его реальности. Если бы мы моделировали мир таким, какой он есть, то атрибут «дата рождения» для объекта Личность следовало бы определить как постоянный. Однако в большинстве случаев может потребоваться его изменение, если обнаружится, что предыдущая запись была ошибочной.

Классификация и обобщение

Мне часто приходится слышать разговоры разработчиков о механизме подтипов как об отношении «является». Я настоятельно рекомендую держаться подальше от такого представления. Проблема заключается в том, что фраза «является» может иметь самый разный смысл.

Рассмотрим следующие фразы:

1. Шеп является Пограничным Колли.
2. Пограничный Колли является Собакой.
3. Собаки являются Животными.
4. Пограничный Колли является Породой Собак.
5. Собака является Биологическим Видом.

Теперь попытаемся скомбинировать эти фразы. Если объединим фразы 1 и 2, то получим «Шеп является Собакой»; фразы 2 и 3 дают в ре-

зультате «Пограничные Колли являются Животными». Объединение первых трех фраз дает «Шеп является Животным». Чем дальше, тем лучше. Теперь попробуем 1 и 4: «Шеп является Породой Собак». Сочетание фраз 2 и 5 дает «Пограничный Колли является Биологическим Видом». Это уже не так хорошо.

Почему некоторые из этих фраз можно комбинировать, а другие нельзя? Потому что некоторые фразы представляют собой **классификацию** (объект Шеп является экземпляром типа Пограничный Колли), а другие – **обобщение** (тип Пограничный Колли является подтипом типа Собака). Обобщение транзитивно, а классификация – нет. Если обобщение следует за классификацией, то в этом направлении их можно комбинировать, а если наоборот, то нельзя.

Смысл сказанного заключается в том, что с отношением «является» следует обращаться весьма осторожно. Его использование может привести к неверному применению подклассов и ошибочным результатам. В приведенном примере хорошими тестами для проверки подтипов могут служить следующие фразы: «Собаки являются разновидностью Животных» и «Каждый экземпляр Пограничного Колли является экземпляром Собаки».

Квалифицированные ассоциации

Квалифицированная ассоциация в языке UML эквивалентна таким известным понятиям в языках программирования, как ассоциативные массивы (associative arrays), схемы (maps) и словари.

На рис. 6.13 показан способ представления ассоциации между Заказом и Строкой Заказа, в котором используется квалификатор. Квалификатор указывает, что в соответствии с Заказом для каждого экземпляра Продукта может существовать одна Строка Заказа.

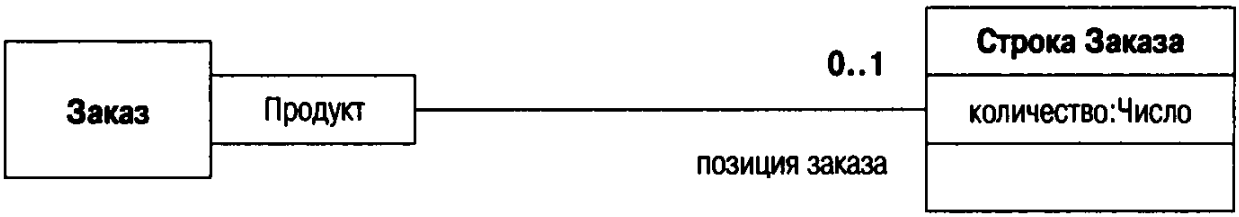


Рис. 6.13. Квалифицированная ассоциация

С концептуальной точки зрения этот пример показывает, что отдельный Заказ не может содержать две Строки Заказа для одного и того же Продукта. С точки зрения спецификации данная квалифицированная ассоциация может повлечь создание интерфейса следующего вида:

```
Class Order {
    public OrderLine getLineItem(Product aProduct);
    public void addLineItem(Number amount, Product forProduct) ;
```

Таким образом, любой доступ к заданной Позиции Заказа требует подстановки некоторого Продукта в качестве аргумента. Кратность «1» означает, что для каждого Продукта должна существовать только одна Позиции Заказа; кратность «*» означает, что для Продукта может существовать несколько Строк Заказа, однако соответствующий доступ к Позициям Заказа все равно выполняется на основе индексов, образованных с помощью Продукта.

С точки зрения реализации для этой цели можно предложить использовать ассоциативный массив или другую аналогичную структуру данных для хранения строк заказа.

```
Class Order {
    private Map _lineItems ;
```

В ходе концептуального моделирования я использую конструкцию квалификатора только для того, чтобы показать ограничения относительно отдельных позиций – «единственная Строка Заказа для каждого Продукта в Заказе». В моделях уровня спецификации нужен квалификатор, чтобы показать наличие интерфейса для поиска по ключу. Лично меня вполне устраивает одновременное использование как квалифицированной, так и неквалифицированной ассоциации, если при этом существует подходящий интерфейс.

В моделях реализации я использую квалификаторы для того, чтобы показать использование схемы, словаря, ассоциативного массива или другой аналогичной структуры данных.

Класс-ассоциация

Классы-ассоциации позволяют дополнительно определять для ассоциаций атрибуты, операции и другие свойства, как это показано на рис. 6.14.

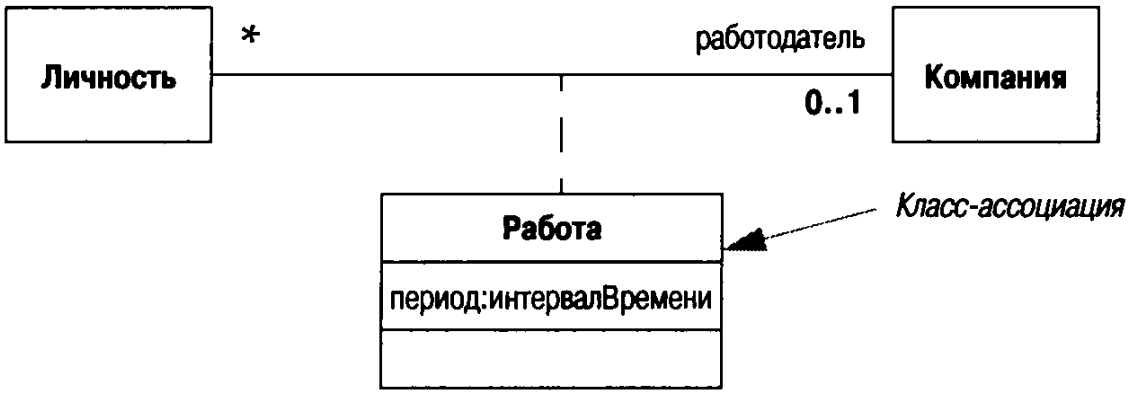


Рис. 6.14. Класс-ассоциация

Из данной диаграммы видно, что Личность может работать только в одной Компании. При этом необходимо каким-то образом хранить информацию относительно периода времени, в течение которого каждый служащий работает в каждой Компании.

Это можно сделать, добавив в данную ассоциацию атрибут «интервалВремени». Можно было бы включить этот атрибут в класс Личность, однако на самом деле он характеризует не Личность, а ее отношение к Компании, которое будет изменяться при смене работодателя.

На рис. 6.15 показан другой способ представления данной информации: преобразование Работы в отдельный самостоятельный класс. (Обратите внимание, как при этом изменили свои значения соответствующие кратности.) В данном примере каждый из классов в первоначальной ассоциации обладал однозначным концом ассоциации по отношению к классу Работа. После преобразования конец ассоциации «работодатель» становится производным, хотя это можно не показывать вовсе.

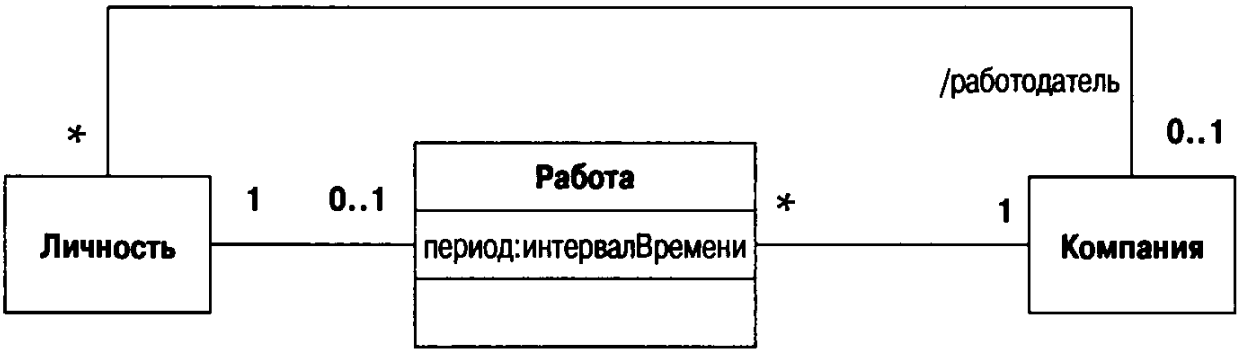


Рис. 6.15. Преобразование класса-ассоциации в обычный класс

Что же полезного может дать класс-ассоциация в качестве компенсации за необходимость помнить еще один вариант уже описанной нотации? Класс-ассоциация дает возможность определить дополнительное ограничение, согласно которому двум участвующим в ассоциации объектам может соответствовать только один экземпляр класса-ассоциации. Мне кажется, что необходимо привести еще один пример.

Посмотрим на две диаграммы, изображенные на рис. 6.16. Форма этих диаграмм практически одинакова. Однако мы можем представить себе некоторую Личность, работающую в одной и той же Компании в различные периоды времени, т. е. он или она увольняется с работы, а позже вновь восстанавливается. Это означает, что Личность в течение некоторого времени может иметь более чем одну ассоциацию Работа с одной и той же Компанией. Что же касается классов Личность и Квалификация, трудно представить себе ситуацию, когда Личность могла бы обладать более чем одним уровнем Компетентности в рамках одной и той же Квалификации. Действительно, такую ситуацию, по всей видимости, следует рассматривать как ошибку.

В языке UML разрешается использовать только второй вариант. Для каждой комбинации Личности и Квалификации может существовать только одна Компетентность.

Верхняя диаграмма на рис. 6.16 не позволяет какой бы то ни было Личности иметь более чем одну Работу в одной и той же Компании. Ес-

ли же необходимо разрешить совмещать различную работу, то Работу следует преобразовать в обычный класс, как это сделано на рис. 6.15.

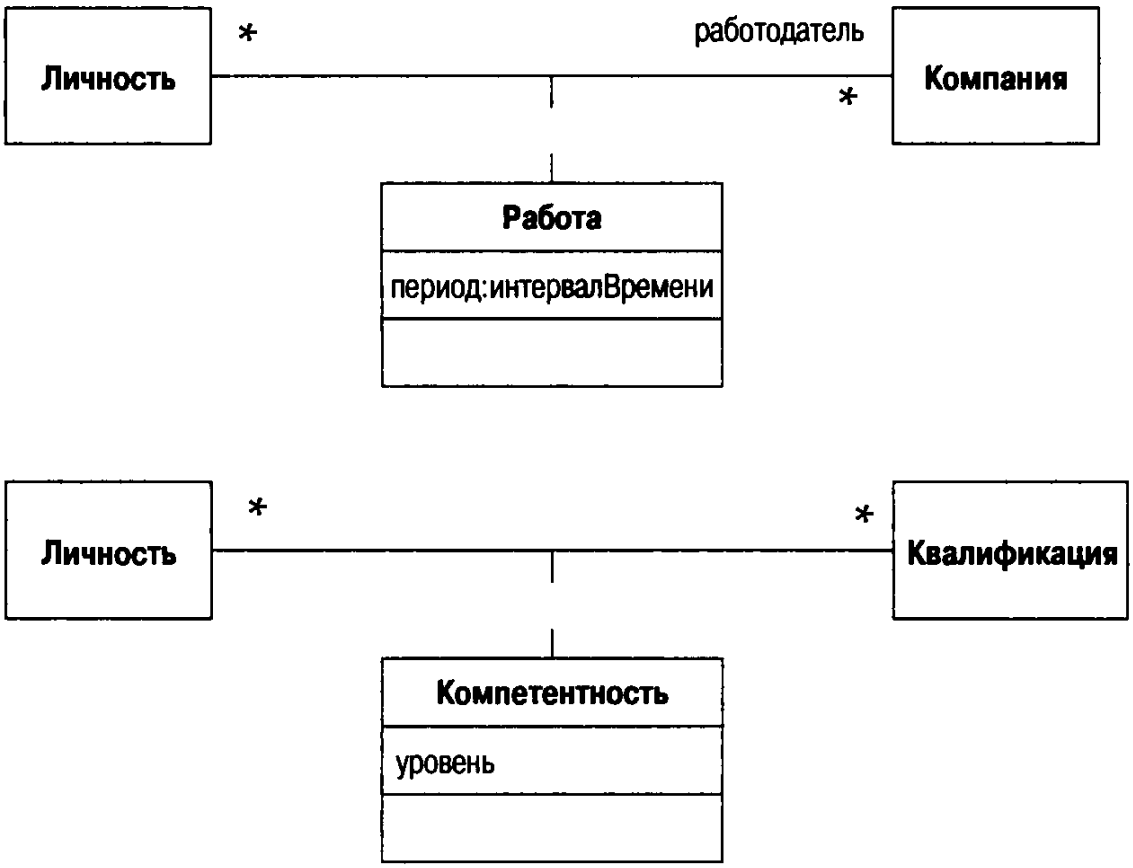


Рис. 6.16. Тонкие особенности класса-ассоциации

Ранее в подобных случаях специалисты по моделированию принимали самые разнообразные допущения относительно смысла класса-ассоциации. Некоторые из них предполагали, что могут существовать только уникальные комбинации, такие как в случае компетентности, в то время как другие избегали какого бы ни было ограничения.

Многие аналитики вовсе не задумывались об этом и могли в одних случаях допускать наличие ограничения, а в других случаях – нет. Таким образом, когда вы пользуетесь языком UML, помните, что данное ограничение присутствует всегда.

Часто этот вид конструкции можно встретить там, где речь идет об исторической информации, как в предыдущем примере с Работой. В подобной ситуации может оказаться полезным образец «Historic Mapping», описанный в книге (Фаулер, 1997 [18]). Мы можем использовать его, определив стереотип «история» («history»), как на рис. 6.17.

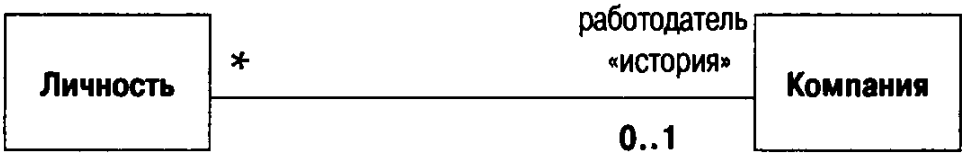


Рис. 6.17. Стереотип «история» для ассоциаций

Данная модель устанавливает, что в каждый момент времени отдельная Личность может работать только в единственной Компании. Однако в течение некоторого периода времени Личность может работать в нескольких компаниях. Это условие предполагает наличие интерфейса следующего вида:

```
class Person {
  //получить текущего работодателя
  Company getEmployer( ) ;
  //работодатель на конкретную дату
  Company getEmployer(Date) ;
  void changeEmployer(Company newEmployer, Date changeDate);
  void leaveEmployer(Date changeDate);
```

Стереотип «история» не является частью языка UML, однако я упомянул о нем здесь по двум причинам. Во-первых, эта нотация в нескольких случаях моделирования оказалась для меня весьма полезной. Во-вторых, она показывает, как можно использовать стереотипы для расширения языка UML.

Параметризованный класс

Некоторые языки, в особенности C++, включают в себя понятие **параметризованного класса** или **шаблона** (template).

Наиболее очевидная польза от применения этого понятия проявляется при работе с некоторыми совокупностями в сильно типизированных языках. Таким образом, в общем случае поведение для множеств можно определить с помощью шаблона класса Множество (Set).

```
class Set <T> {
void insert(T newElement);
void remove(T anElement) ;
```

После этого можно использовать это общее определение для задания конкретных классов-множеств.

```
Set <Employee> employeeSet;
```

Для этой цели в языке UML можно применить параметризованный класс, используя изображенную на рис. 6.18 нотацию.

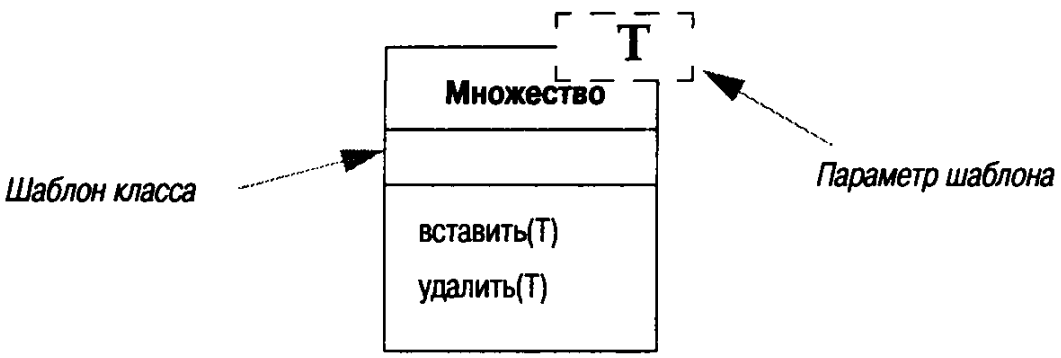


Рис. 6.18. Параметризованный класс

Верхний прямоугольник с буквой T на диаграмме является тем местом, где указывается параметр типа. (Можно указать более одного параметра.) В нетипизированных языках, таких как язык Smalltalk, такой вопрос не возникает, поэтому от данного понятия нет никакой пользы.

Подобное использование параметризованного класса, например *Множество <Служащие>*, называется **связанным элементом** (bound element).

Связанный элемент можно изобразить двумя способами. Первый способ отражает синтаксис языка C++ (рис. 6.19).

Рис. 6.19. *Связанный элемент (версия 1)*

Альтернативная нотация (рис. 6.20) усиливает связь с шаблоном и допускает переименование связанного элемента.

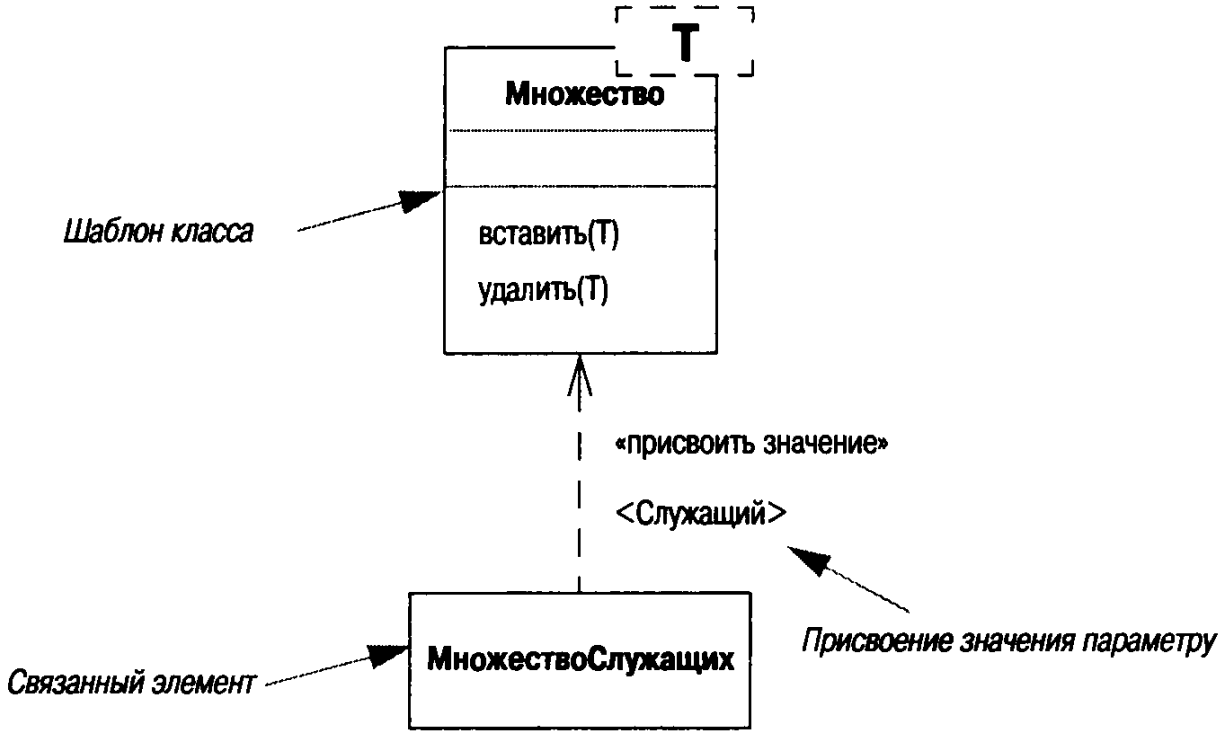


Рис. 6.20. *Связанный элемент (версия 2)*

Стереотип «присвоить значение» («bind») является стереотипом отношения уточнения. Эта отношение показывает, что МножествоСлужащих согласовано с интерфейсом Множества. В терминах спецификации МножествоСлужащих является подтипом Множества. Это соответствует другому способу реализации совокупностей конкретных типов, который заключается в объявлении всех необходимых подтипов.

Однако использование связанного элемента не эквивалентно определению подтипа. В связанный элемент нельзя добавлять свойства – он полностью определяется своим шаблоном. При этом можно добавлять только информацию, ограничивающую его тип. Если же вы хотите добавить свойства, то должны создать некоторый подтип.

Параметризованные классы допускают использование производных типов. Если вы создаете тело шаблона, то можете определить операции над соответствующим параметром. Когда в последующем вы объявите связанный элемент, компилятор попытается установить, поддерживает ли передаваемый параметр операции, определенные в данном шаблоне.

Именно эти действия соответствуют механизму производных типов, поскольку можно вообще не определять тип параметра; компилятор сам вычисляет его в процессе связывания с источником шаблона. Это свойство является центральным при использовании параметризованных классов из библиотеки стандартных шаблонов STL (Standard Template Library) в языке C++; эти классы могут также использоваться для решения других программистских задач.

Использование параметризованных классов не обходится без последствий – например, в языке C++ они могут повлечь за собой значительное увеличение объема кода. Я редко использую параметризованные классы в концептуальном моделировании, поскольку они применяются в основном для совокупностей, которые следует моделировать посредством ассоциаций. (Исключение составляет лишь один случай, когда я все-таки пользуюсь ими. Он связан с образцом «Range» (Диапазон) (см. Фаулер, 1997 [18].) Параметризованные классы бывают мне необходимы только в моделях уровня спецификации и реализации, если они поддерживаются тем языком программирования, на котором я работаю.

Видимость

Должен признаться, что испытываю некоторое беспокойство относительно данного раздела.

Видимость – это одно из тех понятий, которые являются простыми по существу, однако обладают сложными тонкостями. Сама идея видимости заключается в том, что у любого класса имеются общедоступные (public) и закрытые (private) элементы. Общедоступные элементы могут быть использованы любым другим классом, а закрытые элементы – только классом-владельцем. Несмотря на это в каждом языке программирования существуют свои собственные правила. Хотя многие языки используют такие термины, как «общедоступный», «закрытый» и «защищенный» (protected), в разных языках они имеют различное содержание. Эти различия невелики, однако они приводят к

недоразумениям, особенно тех из нас, кто использует в своей работе более одного языка программирования.

Язык UML пытается решить эту проблему, не устраивая при этом жуткую путаницу. По существу, в рамках языка UML для любого атрибута или операции можно указать индикатор видимости. Для этой цели можно использовать любой подходящий маркер, смысл которого определяется тем или иным языком программирования. Однако язык UML предлагает три (довольно трудно запомнить) отдельных обозначения для этих вариантов видимости: «+» (общедоступный), «–» (закрытый) и «#» (защищенный).

Мне бы очень хотелось поскорее закончить на этом, но, к сожалению, разработчики при построении диаграмм используют видимость в своей собственной интерпретации. Поэтому, чтобы действительно понять некоторые из основных различий, существующих между моделями, необходимо понимать трактовку видимости в различных языках программирования. Итак, сделаем глубокий вдох и погрузимся во мрак.

Мы начнем с языка программирования C++, поскольку он является основой стандартного использования языка UML:

- Общедоступный элемент является видимым в любом месте программы и может быть вызван любым объектом в системе.
- Закрытый элемент может быть использован только тем классом, в котором он определен.
- Защищенный элемент может быть использован только а) тем классом, в котором он определен, или б) подклассом этого класса.

Рассмотрим класс Клиент и его подкласс Индивидуальный Клиент. Рассмотрим также объект Мартин, который является экземпляром класса Индивидуальный Клиент. Мартин может использовать любой общедоступный элемент любого объекта в системе. Мартин может также использовать любой закрытый элемент класса Индивидуальный Клиент. Мартин *не* может использовать никакой закрытый элемент, определенный внутри класса Клиент, однако он может использовать защищенные элементы класса Клиент и защищенные элементы класса Индивидуальный Клиент.

Теперь обратимся к языку Smalltalk. В этом языке все переменные экземпляра являются закрытыми, а все операции – общедоступными. Однако закрытость в языке Smalltalk имеет не тот же самый смысл, что в языке C++. В системе, написанной на Smalltalk, Мартин может иметь доступ к любой переменной экземпляра своего собственного объекта, независимо от того, где была определена эта переменная экземпляра: в Клиенте или в Индивидуальном Клиенте. Таким образом, закрытость в языке Smalltalk имеет тот же смысл, что и защищенность в языке C++.

Тем не менее, было бы слишком просто закончить на этом.

Вернемся опять к языку C++. Пусть имеется еще один экземпляр класса Индивидуальный Клиент с именем Кендалл. Кендалл может иметь доступ к любому элементу Мартина, который был определен как часть класса Индивидуальный Клиент, независимо от того, является ли он общедоступным, закрытым или защищенным. Кендалл может также иметь доступ к любому защищенному или общедоступному элементу Мартина, который был определен в классе Клиент. Однако в языке Smalltalk Кендалл не может получить доступ к закрытым переменным экземпляра Мартина, а только к общедоступным операциям Мартина.

В языке C++ доступ к элементам других объектов вашего собственного класса обеспечивается в той же степени, что и к вашим собственным элементам. В языке Smalltalk безразлично, принадлежит ли другой объект к тому же самому классу или нет; вам все равно доступны только общедоступные элементы другого объекта.

Язык Java похож на язык C++ в том, что он поддерживает свободный доступ к элементам других объектов одного и того же класса. В языке Java введен дополнительный уровень видимости, получивший название «пакет» (package). Элемент с видимостью внутри пакета может быть доступен только в экземплярах других классов этого же пакета.

Продолжая эту тему, следует отметить, что все это вовсе не так просто; в языке Java несколько по иному определяется защищенная видимость. В Java защищенный элемент может быть доступен не только подклассам, но и любому другому классу того же самого пакета, к которому относится класс-владелец. Это означает, что в языке Java защищенная видимость является более общедоступной, чем пакетная.

Язык Java разрешает также помечать классы как общедоступные или пакетные. Общедоступные элементы общедоступного класса могут использоваться любым классом, который импортирует пакет, содержащий исходный класс. Пакетный класс может быть использован только другими классами в том же самом пакете.

Последний штрих в эти тонкости добавляет язык C++. В языке C++ какой-либо метод или класс может быть определен как «дружественный» (friend) для класса. Такой **дружественный элемент** обладает полным доступом ко всем элементам класса. Отсюда пошло высказывание: «в C++ друзья прикасаются к закрытым частям друг друга».

При определении видимости пользуйтесь правилами того языка, на котором вы программируете. С какой бы точки зрения вы не смотрели на модель UML, следует с осторожностью относиться к смыслу маркеров видимости и осознавать, что их смысл может меняться от языка к языку.

По моему мнению, обычно видимость изменяется в процессе кодирования. Поэтому не следует определять ее слишком рано.

7

Пакеты и кооперации

Один из старейших вопросов методологии разработки программного обеспечения: как разбить большую систему на небольшие подсистемы? Мы задаем этот вопрос, поскольку чем больше становятся системы, тем труднее разбираться в них и во вносимых в них изменениях.

Структурные методы использовали **функциональную декомпозицию**, согласно которой вся система в целом представляется как одна функция и разбивается на подфункции, которые в свою очередь тоже разбиваются на подфункции и т. д. Эти функции похожи на варианты использования в объектно-ориентированной системе в том, что функции представляют собой действия, выполняемые системой в целом.

В прежние времена процесс и данные рассматривались отдельно друг от друга. Другими словами, помимо функциональной декомпозиции существовала также структура данных. Она имела второстепенное значение, хотя некоторые методы информационных технологий группировали записи данных в предметные области и формировали матрицы, чтобы показать взаимосвязь функций и записей данных.

Именно с этой точки зрения мы можем оценить те огромные изменения, которые произвели объекты. Разделение процесса и данных осталось в прошлом, функциональная декомпозиция тоже, однако старейший вопрос по-прежнему все еще существует.

Одна из идей заключается в группировке классов в блоки более высокого уровня. Эта идея, применяемая самым произвольным образом,

проявляется во многих объектных методах. В языке UML такой механизм группировки получил название **пакет** (package).

Пакеты

Идея пакета может быть применена не только к классам, но и к любому другому элементу модели. Без некоторых эвристических процедур группировка классов может оказаться произвольной. Одной из них, которую я считаю самой полезной и которая повсеместно используется в языке UML, является зависимость.

Я использую термин **диаграмма пакетов** по отношению к диаграмме, которая содержит пакеты классов и зависимости между ними. Строго говоря, диаграмма пакетов является всего лишь разновидностью диаграммы классов, на которой показаны только пакеты и зависимости. Сам я использую эти диаграммы очень часто, поэтому и называю их диаграммами пакетов. Однако следует помнить, что этот термин введен мною, а не является официальным именем диаграммы в языке UML.

Между двумя элементами существует **зависимость**, если изменения в определении одного элемента могут повлечь за собой изменения в другом. Что касается классов, то характер зависимостей может быть самым различным: один класс посылает сообщение другому; один класс включает другой класс как часть своих данных; один класс ссылается на другой как на параметр некоторой операции. Если класс меняет свой интерфейс, то любое сообщение, которое он посылает, может оказаться ошибочным.

В идеальном случае только изменения в интерфейсе класса должны оказывать влияние на другие классы. Искусство проектирования больших систем включает в себя минимизацию зависимостей; таким образом влияние изменений уменьшается, а для изменения системы требуются меньшие усилия.

В языке UML имеются разнообразные виды зависимостей, каждый из которых обладает самостоятельной семантикой и стереотипом. По моему мнению, намного проще начинать с зависимости без стереотипа и использовать более конкретные виды зависимостей только по мере необходимости.

На рис. 7.1 изображены классы предметной области, моделирующие бизнес-систему и сгруппированные в два пакета: Заказы и Клиенты.

Оба пакета являются частью пакета предметной области в целом. Приложение Сбора Заказов имеет зависимости с обоими пакетами предметной области. Пользовательский Интерфейс Сбора Заказов имеет зависимости с Приложением Сбора Заказов и AWT (средством разработки графического интерфейса пользователя в языке Java).

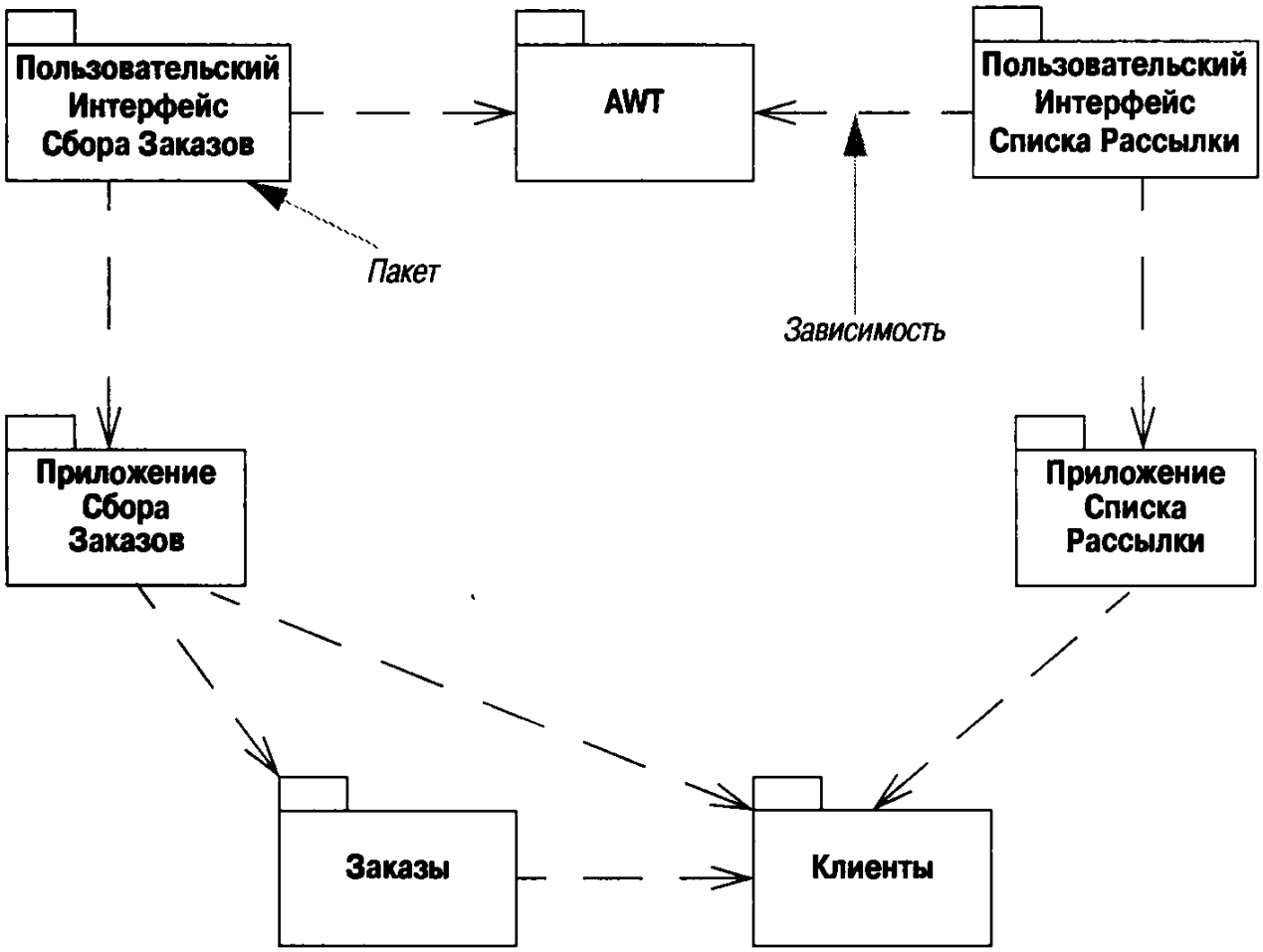


Рис. 7.1. Диаграмма пакетов

Между двумя пакетами существует некоторая зависимость, если существует какая-либо зависимость между любыми двумя классами в пакетах. Например, если любой класс в пакете Список Рассылки зависит от какого-либо класса в пакете Клиенты, то между этими пакетами существует зависимость.

Существует очевидное сходство между зависимостями пакетов и зависимостями компиляции. Тем не менее, между ними имеется принципиальное различие: зависимости пакетов не являются транзитивными.

Можно привести следующий пример **транзитивного** отношения: у Джима борода длиннее, чем у Гради, а у Гради длиннее, чем у Айвара, отсюда можно заключить, что у Джима борода длиннее, чем у Айвара.

Чтобы понять, почему это так важно для зависимостей, обратимся снова к рис. 7.1. Если изменяется какой-либо класс в пакете Заказы, то это совсем не означает, что должны быть внесены изменения в пакет Пользовательский Интерфейс Сбора Заказов. Это всего лишь означает, что нужно проверить, не изменился ли пакет Приложение Сбора Заказов. Пакет Пользовательский Интерфейс Сбора Заказов может потребовать изменений только в том случае, если изменится интерфейс пакета Приложение Сбора Заказов. В данной ситуации Приложение Сбора Заказов защищает Пользовательский Интерфейс Сбора Заказов от изменений в заказах.

Такое поведение системы является классической особенностью многоуровневой архитектуры. Действительно, именно такова семантика поведения конструкции «imports» в языке Java, но поведение конструкции «includes» в языке C/C++ другое. В языке C/C++ конструкция «includes» является транзитивной, а это означает, что Пользовательский Интерфейс Сбора Заказов следует считать зависимым от пакета Заказы. Транзитивная зависимость затрудняет ограничение области действия изменений при компиляции. (Хотя большинство зависимостей не являются транзитивными, вы можете определить специальный стереотип для этой цели.)

Классы в пакетах могут быть общедоступными, закрытыми и защищенными. Таким образом, пакет Заказы зависит от общедоступных методов общедоступного класса в пакете Клиенты. Если изменить некоторый закрытый метод в любом классе пакета Клиенты или общедоступный метод в каком-либо закрытом классе пакета Клиенты, то эти изменения не затронут ни один из классов в пакете Заказы.

В данном случае может оказаться полезным сократить интерфейс этого пакета за счет экспорта только небольшого подмножества операций, ассоциированных с общедоступными классами в этом пакете. Это можно сделать присвоением всем классам закрытой видимости с тем, чтобы они могли быть видимы только для других классов того же самого пакета, а также посредством добавления экстрадоступных классов для общедоступного поведения. После чего эти экстра-классы, получившие название *фасадов* (*facades*) (Гамма и др., 1995 [20]), делегируют общедоступные операции своим соседям по пакету.

Хотя пакеты не дают ответа на вопрос, как уменьшить количество зависимостей в вашей системе, однако они помогают выделить эти зависимости. Как только они окажутся на виду, вам останется лишь поработать над их сокращением. По моему мнению, диаграммы пакетов являются основным средством управления общей структурой системы.

На рис. 7.2 изображена более сложная диаграмма пакетов, содержащая ряд дополнительных конструкций.

Во-первых, мы видим, что добавлен пакет Предметная Область, который содержит пакеты Заказы и Клиенты. Это представляется весьма полезным, поскольку означает, что вместо множества отдельных зависимостей можно изобразить зависимости, направленные к этому пакету и от пакета в целом.

Когда показывается содержимое некоторого пакета, то имя пакета помещается в небольшой верхний прямоугольник, а состав пакета изображается внутри основного прямоугольника. Пакет может содержать внутри себя перечень некоторых классов, как в случае пакета Общий; другую диаграмму пакетов, как в случае пакета Предметная Область; или некоторую диаграмму классов (которая не показана, но идея сама по себе является очевидной).

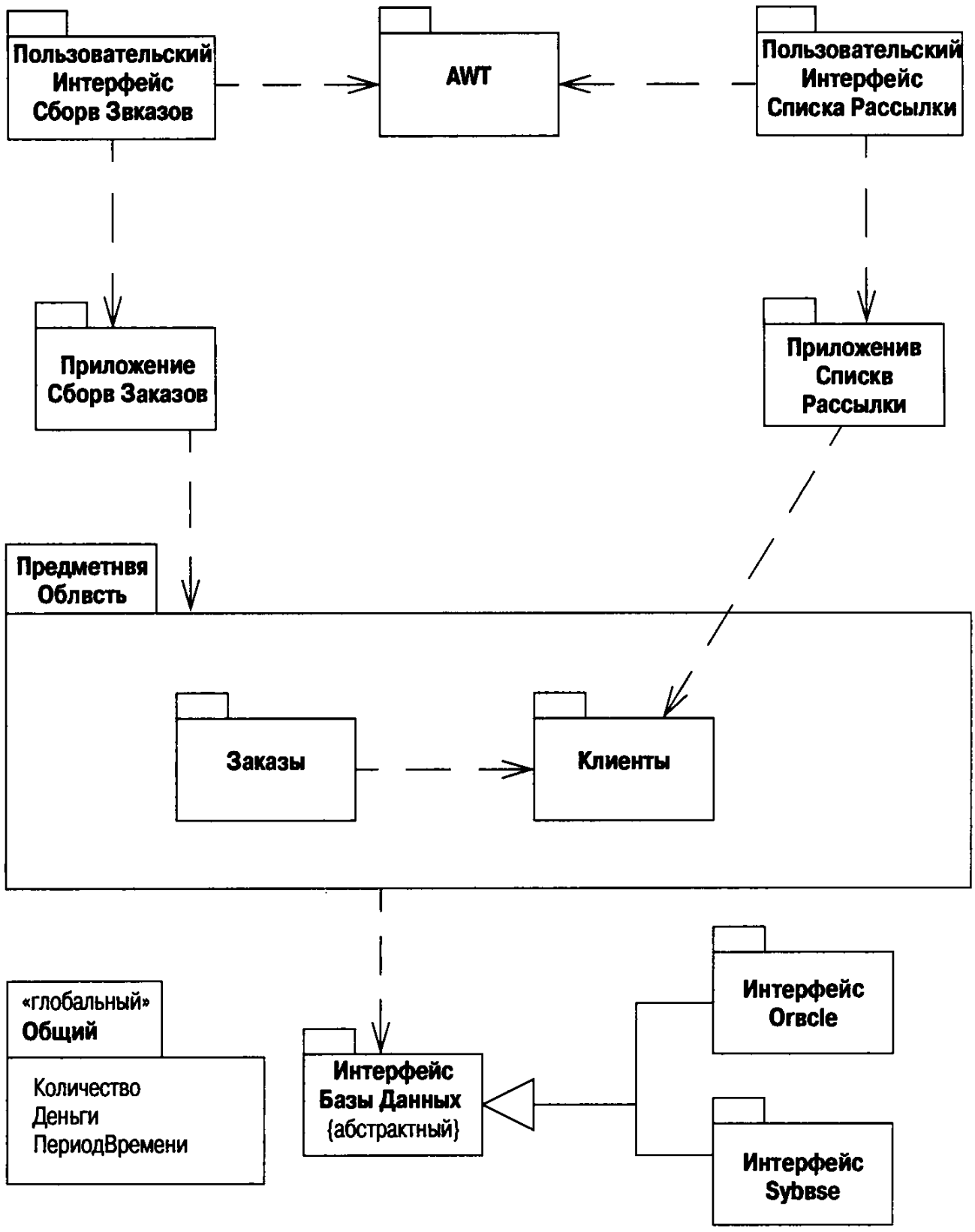


Рис. 7.2. Расширенная диаграмма пакетов

Я считаю, что в большинстве случаев вполне достаточно перечислить основные классы, но иногда оказывается полезным указать на диаграмме дополнительную информацию. В данном случае я показал, что хотя Приложение Сбора Заказов связано зависимостью со всем пакетом Предметная Область, Приложение Списка Рассылки зависит только от пакета Клиенты.

Что означает зависимость, изображенная по направлению к пакету, который в свою очередь содержит другие пакеты? В общем случае это означает, что такая зависимость обобщает зависимость более низкого уровня. Другими словами, если существует некоторая зависимость от некоторого элемента, содержащегося внутри пакета более высокого уровня, то для представления подобных деталей необходима более подробная диаграмма. Кроме того, точные правила изменяются для каждого отдельного множества зависимостей.

На рис. 7.2 изображен пакет Общий, помеченный как «глобальный». Это означает, что все пакеты в системе зависят от данного пакета. Очевидно, такую конструкцию следует применять весьма осторожно, однако некоторые общие классы, такие как Деньги, используются всеми элементами системы.

К пакетам можно применять отношение обобщения. Это означает, что более частный пакет должен быть согласован с интерфейсом общего пакета. Именно такое определение сопоставимо с точкой зрения спецификации на механизм подклассов в диаграммах классов (см. главу 4). Следовательно, в соответствии с рис. 7.2 Брокер Базы Данных может использовать либо Интерфейс Oracle, либо Интерфейс Sybase. Если обобщение применяется подобным образом, то общий пакет можно пометить как {абстрактный}. Это говорит о том, что общий пакет всего лишь определяет интерфейс, реализуемый более частным пакетом.

Обобщение означает наличие зависимости от подтипа к супертипу. (Нет необходимости дополнительно показывать подобную зависимость; для этого достаточно самого обобщения.) Размещение абстрактных классов внутри пакета-супертипа является хорошим способом избежать появления циклов в структуре зависимостей. В нашем случае пакеты интерфейса базы данных отвечают за загрузку и сохранение объектов предметной области в базе данных. Следовательно, они должны располагать информацией относительно объектов предметной области. Однако инициировать загрузку и сохранение должны сами объекты предметной области.

Обобщение позволяет поместить необходимый интерфейс триггера (различные операции загрузки и сохранения) в пакет интерфейса базы данных. После чего эти операции могут быть реализованы классами в рамках пакетов-подтипов. Таким образом, нет никакой необходимости указывать зависимость между пакетом интерфейса базы данных и пакетом интерфейса Oracle, поскольку во время выполнения объекты предметной области сами будут вызывать соответствующий пакет-подтип. Но сами объекты предметной области будут думать, что имеют дело только с исходным пакетом интерфейса базы данных. Полиморфизм столь же полезен для пакетов, как и для классов.

Как и любая эвристика, удаление циклов из структуры зависимостей является достаточно хорошей идеей. Я не уверен, что можно избавиться от всех циклов, однако следует стремиться минимизировать их ко-

личество. Если у вас имеются подобные циклы, попытайтесь поместить их в более крупный пакет-контейнер. В моей практике встречались случаи, когда я был не способен избавиться от циклов между пакетами предметной области, однако старался исключить их из взаимодействий между интерфейсами предметной области и внешними интерфейсами. Основным средством для исключения циклов является механизм обобщения для пакетов.

В существующей системе зависимости могут быть выведены на основании анализа классов. Эта задача является крайне полезной для реализации с помощью любого инструментального средства. Я всегда пользуюсь такими средствами, когда занимаюсь улучшением структуры существующей системы. Было бы неплохо в качестве самого первого шага сгруппировать классы в пакеты и проанализировать зависимости между пакетами. После чего я прибегаю к реорганизации, чтобы уменьшить количество зависимостей.

Кооперации

Так же как классы-контейнеры, пакет может содержать кооперации. **Кооперация** представляет собой имя, которое присваивается взаимодействию двух и более классов. Обычно подобное взаимодействие изображается на диаграмме взаимодействия. Так, например, диаграмма последовательности на рис. 7.3 представляет кооперацию Организация Продажи.

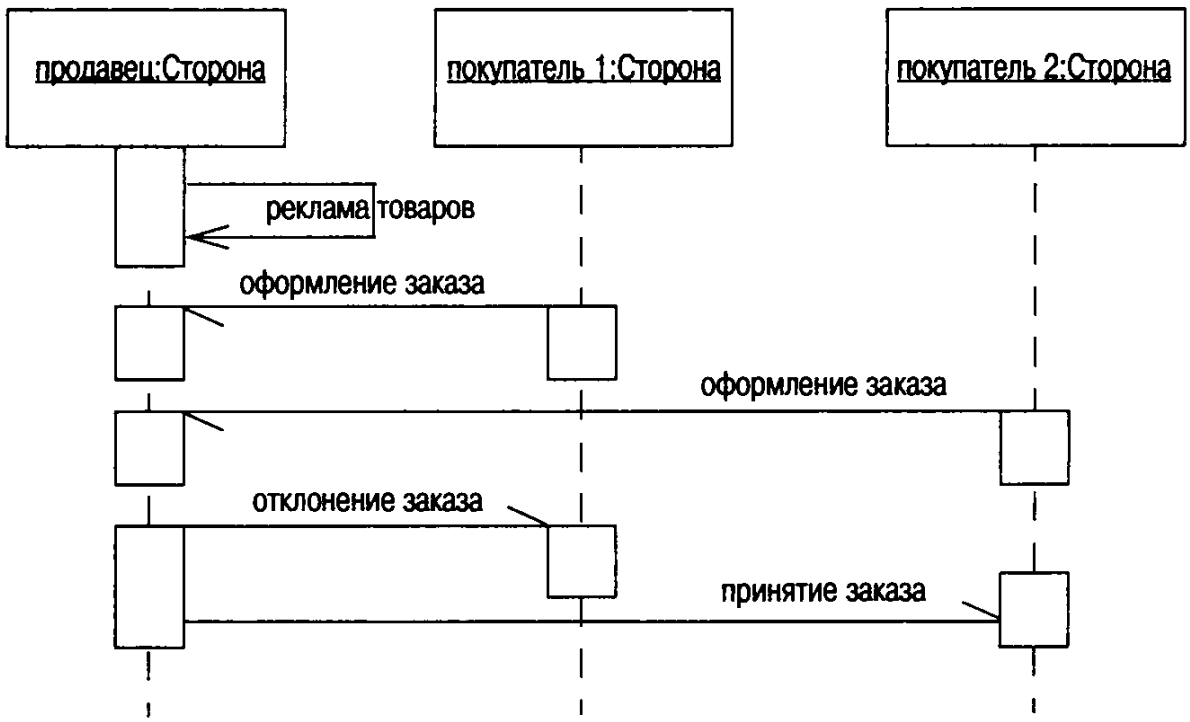


Рис. 7.3. Диаграмма последовательности для организации продажи

Эта кооперация может показывать выполнение операции или реализацию варианта использования. Кооперации можно моделировать до решения о том, какие операции они будут в себя включать.

Кооперация может быть описана более чем одной диаграммой взаимодействия, на каждой из которых показывается отдельная линия поведения. Можно также добавить диаграмму классов, чтобы показать классы, участвующие в заданной кооперации (рис. 7.4).

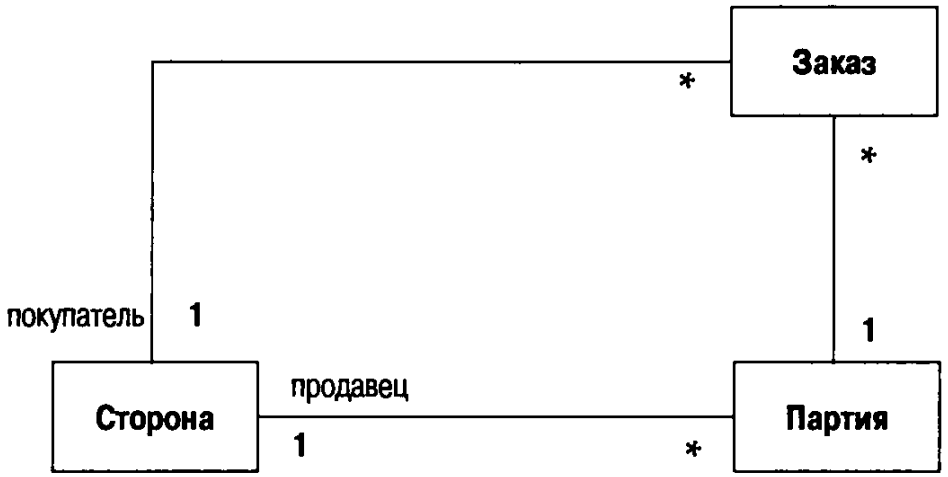


Рис. 7.4. Диаграмма классов для кооперации организации продажи

В дополнение к использованию кооперации в пакете кооперации можно применять для представления общего поведения совокупности пакетов. Если задать вопрос о взаимодействии базы данных, то можно в качестве ответа изобразить несколько пакетов и классов с взаимосвязями между ними. Такое представление поможет понять, как все это работает, хотя взаимодействие базы данных может оказаться лишь одним из аспектов этих пакетов и классов. Можно усовершенствовать это представление посредством определения кооперации взаимодействия базы данных. В рамках этой кооперации можно изобразить только релевантные аспекты классов, а диаграммы взаимодействия покажут, как они работают.

Часто можно столкнуться с ситуацией, при которой одна и та же кооперация используется различными классами в системе. При этом в каждый момент времени основные особенности поведения этих классов идентичны, но классы и операции имеют различные имена, и могут существовать лишь небольшие различия в их реализации. Данную ситуацию можно представить посредством параметризованной кооперации.

Сначала рисуется диаграмма, подобная рис. 7.5, где показаны разные роли, которые исполняют различные объекты в данной кооперации. (Заметим, что классы на этой диаграмме не являются реальными классами системы; в действительности они являются ролями этой кооперации.) Затем следует добавить диаграммы взаимодействия, чтобы показать особенности взаимодействия этих ролей.

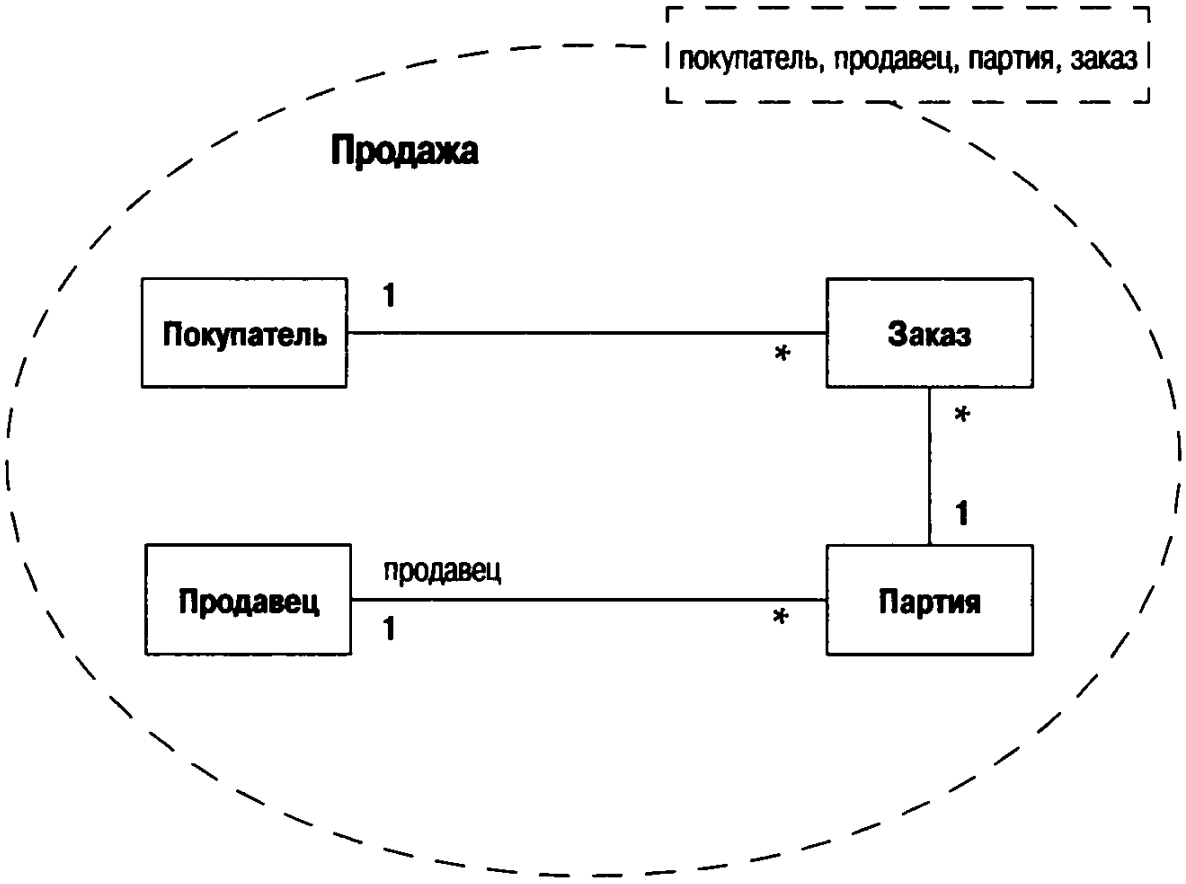


Рис. 7.5. Параметризованная кооперация Продажа

Далее можно показать, как множество классов участвует в этой кооперации, нарисовав диаграмму, подобную рис. 7.6.

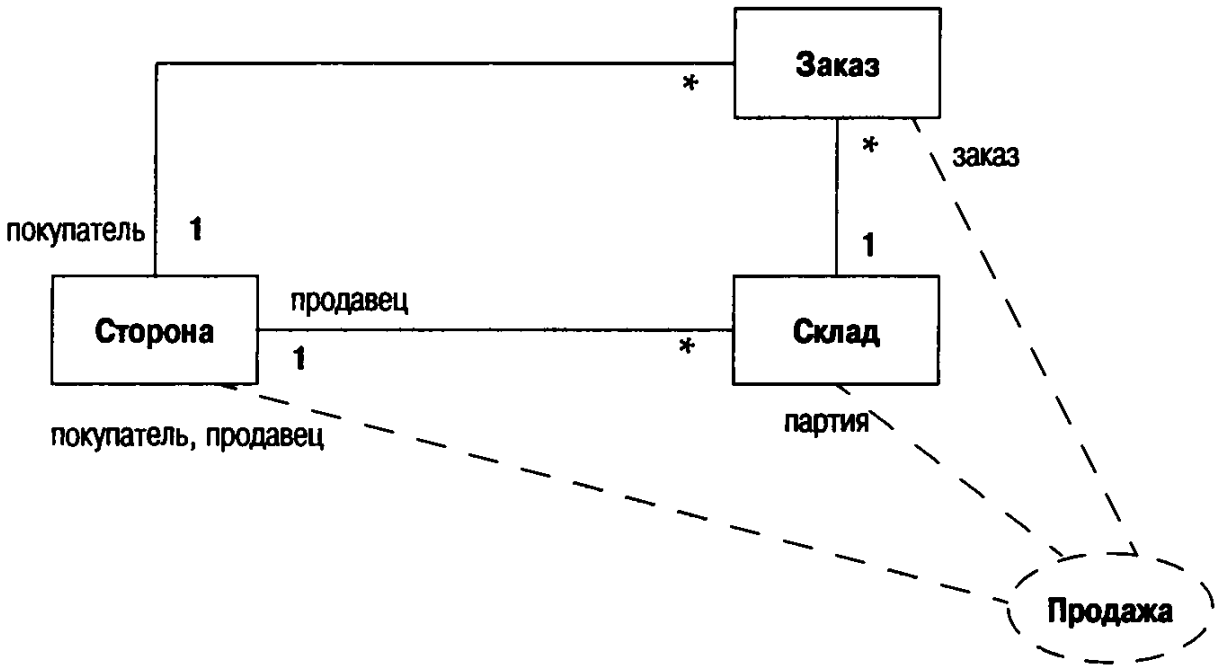


Рис. 7.6. Использование кооперации Продажа

В языке UML используется также термин **образец** (pattern) как синоним параметризованной кооперации. Довольно спорно называть образцом подобную ситуацию, поскольку здесь присутствуют по сути бо-

лее одного образца. Но, без сомнения, эта нотация может быть применена для изображения ситуации, когда в конкретной системе используются общие образцы.

Этот вид нотации может быть использован также в процессе моделирования на основе ролей, в рамках которого вначале моделируются кооперации и роли, а затем разрабатываются классы, которые реализуют эти роли. Дополнительную информацию об этом стиле проектирования можно найти в книге Ринскауга (Reenskaug), 1996 [35].

Когда использовать диаграммы пакетов и кооперации

Пакеты являются чрезвычайно важным средством для больших проектов. Пакеты следует использовать всякий раз, когда диаграмма классов для системы в целом, размещенная на единственном листе бумаги формата A4, становится трудно читаемой.

Пакеты особенно полезны для тестирования. Хотя я и писал некоторые тесты, основываясь исключительно на классах, все же предпочитаю формировать тестовые блоки на основе пакетов. При этом каждый пакет может содержать один или несколько тестовых классов, с помощью которых проверяется поведение этого пакета.

По моему опыту кооперации полезны всякий раз, когда необходимо сослаться на конкретное взаимодействие. Параметризованные кооперации полезны, если в вашей системе имеется несколько похожих кооперации.

Где найти дополнительную информацию

Пакеты подробно рассмотрены в работе Роберта Мартина (Robert Martin), 1995 [30], которая, по моему мнению, является лучшей из книг по данной тематике на сегодняшний день. В этой книге приведен ряд примеров использования метода Буча в сочетании с языком C++, при этом большое внимание уделено минимизации зависимостей. Полезную информацию можно найти также в книге Вирс-Брока (Wirfs-Brock), 1990 [46], в которой автор впервые говорит о пакетах как о подсистемах.

Кооперации являются сравнительно новой темой, поэтому дополнительную информацию можно найти только в более обстоятельных книгах по языку UML.

8

Диаграммы состояний

Диаграммы состояний являются хорошо известным методом описания поведения систем. Они изображают все возможные состояния, в которых может находиться конкретный объект, а также изменения состояния объекта, которые происходят в результате влияния некоторых событий на этот объект. В большинстве объектно-ориентированных методов диаграммы состояний строятся для единственного класса, чтобы показать динамику поведения единственного объекта.

Существует несколько разновидностей представления диаграмм состояний, незначительно отличающихся друг от друга семантикой. Стиль, принятый в языке UML, основан на схемах состояний Дэвида Харела (David Harel), 1987 [22].

На рис. 8.1 изображена диаграмма состояний в обозначениях языка UML, описывающая поведение заказа в системе обработки заказов, которая была рассмотрена ранее в данной книге. На диаграмме представлены различные состояния, в которых может находиться заказ.

Из начальной точки процесс переходит в состояние Проверка. Этот переход имеет метку «/получить первую позицию заказа».

Синтаксис метки перехода состоит из трех частей, каждая из которых является необязательной: *Событие* [*Сторожевое условие*] / *Действие*. В данном случае метка состоит только из действия «получить первую позицию заказа». После выполнения этого действия мы попадаем в состояние Проверка. С этим состоянием ассоциируется некоторая деятельность, которая обозначается меткой со следующим синтаксисом:

выполнить/деятельность. В данном случае деятельность называется «проверить позицию заказа».

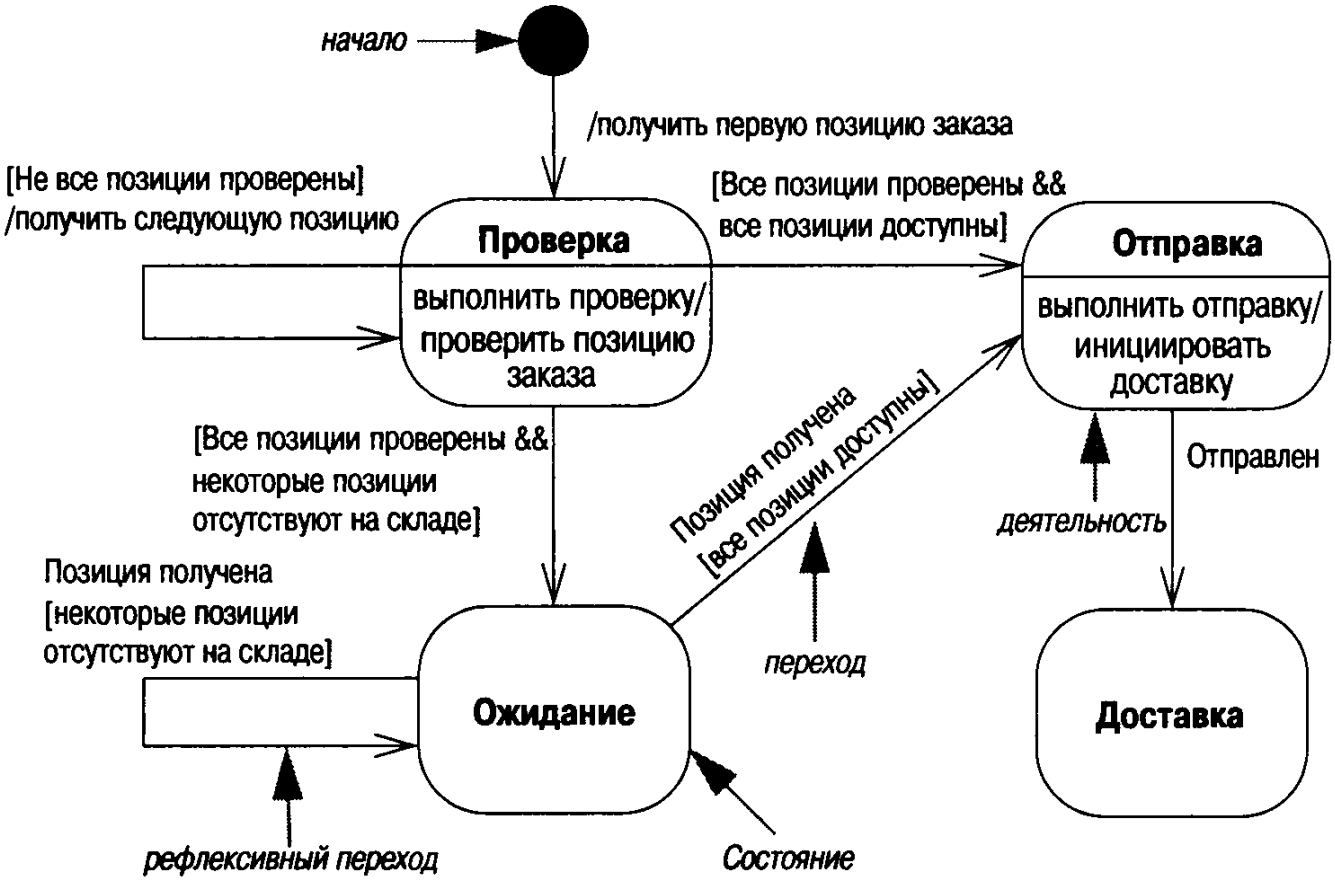


***Рис. 8.1.** Диаграмма состояний*

Следует отметить, что я пользуюсь терминами «действие» (action) для перехода и «деятельность» (activity) для состояния. Хотя оба термина обозначают процессы, обычно реализуемые некоторым методом класса Заказ, они трактуются различным образом. **Действия** ассоциируются с переходами и рассматриваются как мгновенные и непрерываемые. **Деятельности** ассоциируются с состояниями и могут продолжаться достаточно долго. Деятельность может быть прервана некоторым событием.

Обратите внимание, что смысл определения «мгновенный» зависит от типа разрабатываемой системы. Для компьютерных систем реального времени это может соответствовать нескольким машинным командам; для обычной информационной системы «мгновение» может означать менее нескольких секунд.

Если метка перехода не содержит никакого события, это означает, что переход произойдет, как только завершится какая-либо деятельность, ассоциированная с данным состоянием; в данном случае – как только будет выполнена Проверка. Из состояния Проверка выходят три перехода. Метка каждого из них включает только Сторожевое условие. **Сторожевое условие** – это логическое условие, которое может принимать одно из двух значений: «истина» или «ложь». Переход со сторо-

жевым условием выполняется только в том случае, если данное сторожевое условие принимает значение «истина».

Из конкретного состояния в данный момент времени может быть осуществлен только один переход, таким образом, сторожевые условия должны быть взаимно исключающими для любого события. На рис. 8.1 мы имеем дело с тремя условиями:

1. Если проверены не все позиции, входящие в заказ, мы получаем следующую позицию и возвращаемся в состояние Проверка.
2. Если проверены все позиции и все они имеются на складе, то мы переходим в состояние Отправка.
3. Если проверены все позиции, но не все из них имеются на складе, то мы переходим в состояние Ожидание.

Сначала рассмотрим состояние Ожидание. В этом состоянии не существует деятельностей, поэтому данный заказ находится в состоянии ожидания, пока не наступит некоторое событие. Оба перехода из состояния Ожидание помечены событием «Позиция получена». Это означает, что соответствующий заказ находится в состоянии Ожидание до тех пор, пока он не обнаружит наступление данного события. В этот момент оцениваются сторожевые условия данных переходов, и выполняется соответствующий переход либо в состояние Отправка, либо обратно в состояние Ожидание.

В состоянии Отправка имеется деятельность, которая инициирует доставку. Из этого состояния имеется единственный безусловный переход, который происходит в результате наступления события «Отправлен». Это означает, что рассматриваемый переход обязательно произойдет, если наступит данное событие. При этом следует заметить, что этот переход *не* произойдет, даже если завершится деятельность; наоборот, когда деятельность «инициировать доставку» завершится, данный заказ останется в состоянии Отправка, пока не наступит событие «Отправлен».

Наконец, рассмотрим переход с именем «отмена». Мы должны располагать возможностью отменить заказ в любой момент, пока заказ не доставлен клиенту. Это можно сделать, изобразив отдельные переходы из каждого состояния: Проверка, Ожидание и Отправка. Удобный альтернативный вариант – определить некоторое **суперсостояние** для трех перечисленных состояний, после чего нарисовать единственный выходящий из него переход. В этом случае подсостояния просто наследуют любые переходы суперсостояния.

Оба подхода изображены на рис. 8.2 и 8.3. Они описывают одно и то же поведение системы.

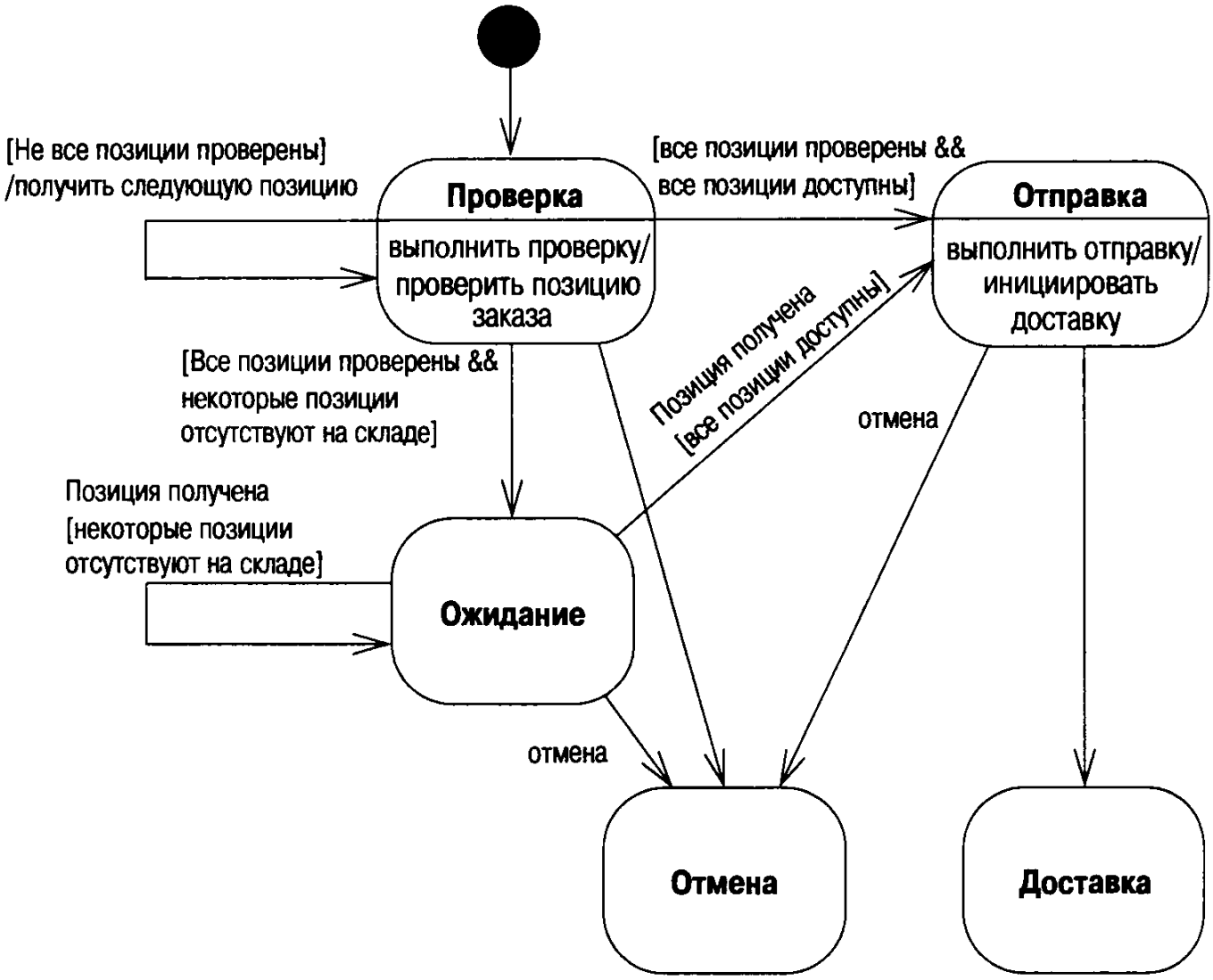


Рис. 8.2. Диаграмма состояний без суперсостояний

Рис. 8.2 выглядит довольно перегруженным, хотя на нем изображено всего три дублирующих перехода. На рис. 8.3 картина в целом выглядит гораздо яснее, и если впоследствии потребуется внести какие-либо изменения, то будет значительно труднее упустить из вида событие «отмена».

В данных примерах я изобразил деятельность внутри состояния в виде текста *«выполнить/деятельность»*. Внутри состояния также можно указать и другую информацию.

Если состояние реагирует на событие, связанное с действием, которое не влечет за собой никакой переход, этот факт можно изобразить, поместив текст вида *«ИмяСобытия / ИмяДействия»* в прямоугольник состояния.

Помимо событий с именами существуют еще два других типа событий:

- Событие может быть инициировано после завершения определенного периода времени. Такое событие можно пометить ключевым словом **после**. Например, можно записать *после (20 минут)*.

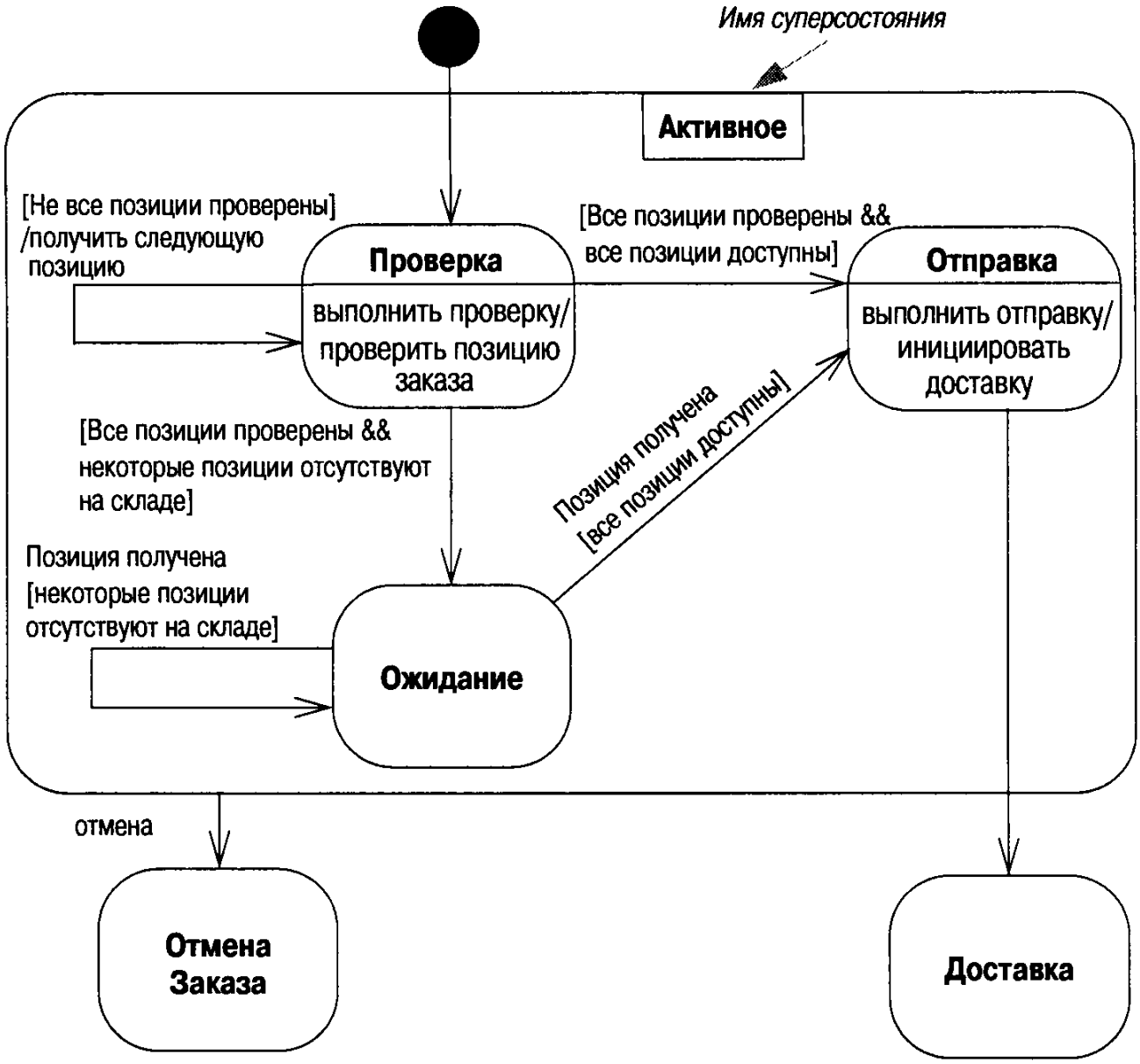


Рис. 8.3. *Диаграмма состояний с суперсостояниями.*

- Событие может быть инициировано в результате выполнения того или иного логического условия. Такое событие можно пометить ключевым словом **если**. Например, можно записать *если (температура >100 градусов)*.

Существуют также два особых события: вход и выход. Любое действие, связанное с событием **входа**, выполняется в момент перехода объекта в данное состояние. Действие, ассоциированное с событием **выхода**, выполняется в том случае, когда объект покидает данное состояние в результате осуществления некоторого перехода. Если имеется так называемый **рефлексивный переход**, возвращающий объект обратно в то же самое состояние и связанный с каким-либо действием, то сначала должно выполниться действие выхода, затем действие данного перехода и, наконец, действие входа. Если с данным состоянием ассоциирована некоторая деятельность, то она начнет выполняться сразу после действия входа.

Диаграммы параллельных состояний

Помимо состояний заказа, связанных с наличием позиций заказа, существуют также состояния, связанные с подтверждением оплаты заказа. Эти состояния могут быть представлены диаграммой состояний, подобной той, которая изображена на рис. 8.4.

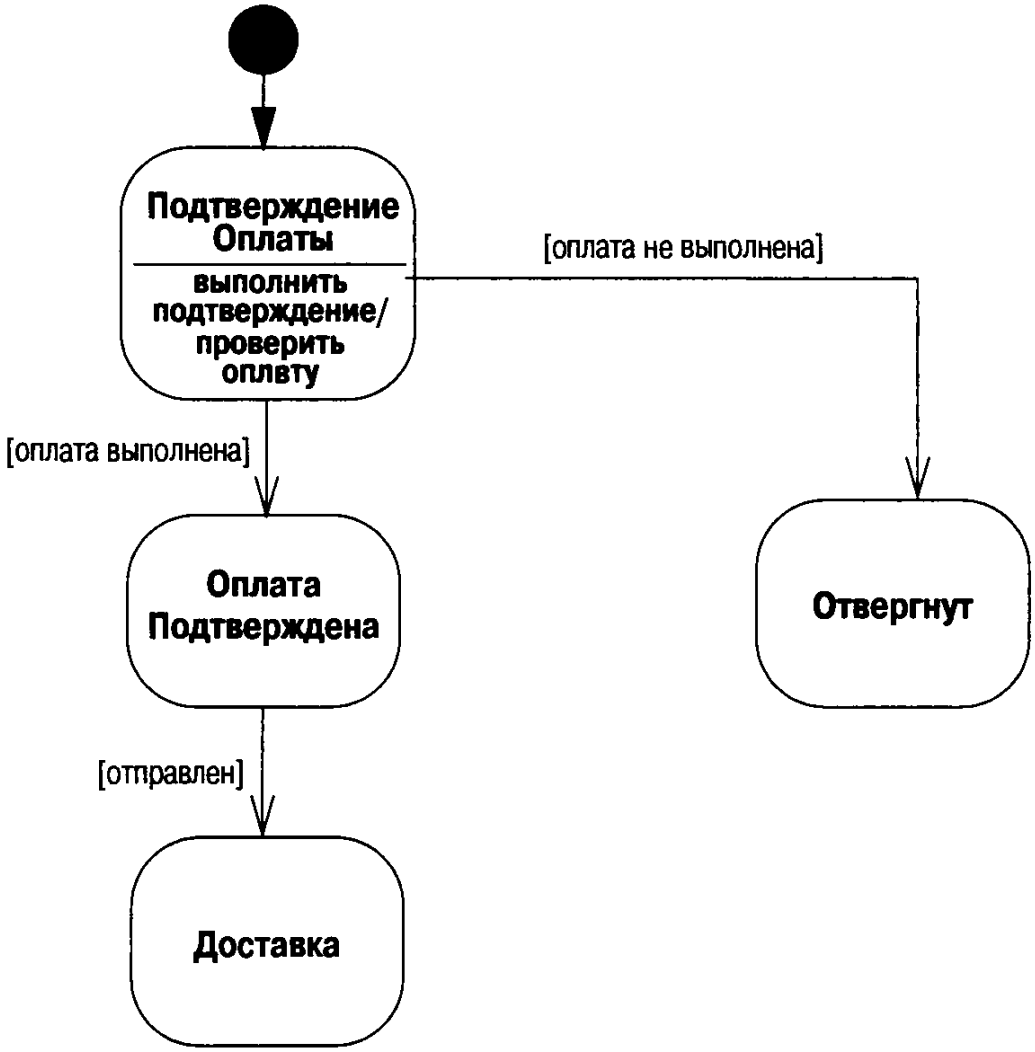


Рис. 8.4. Подтверждение оплаты заказа

В данном случае все начинается с проверки подтверждения оплаты. Деятельность «проверить оплату» завершается сообщением о результате выполнения данной проверки. Если оплата заказа выполнена, то данный заказ ожидает в состоянии Оплата Подтверждена до тех пор, пока не наступит событие «отправлен». В противном случае заказ переходит в состояние Отвергнут.

Таким образом, поведение объекта Заказ определяется как комбинация поведений, изображенных на рис. 8.1 и 8.4. Все эти состояния и рассмотренное ранее состояние Отмена можно объединить в одну **диаграмму параллельных состояний** (рис. 8.5).

Обратите внимание, что на рис. 8.5 детали внутренних состояний не изображены.

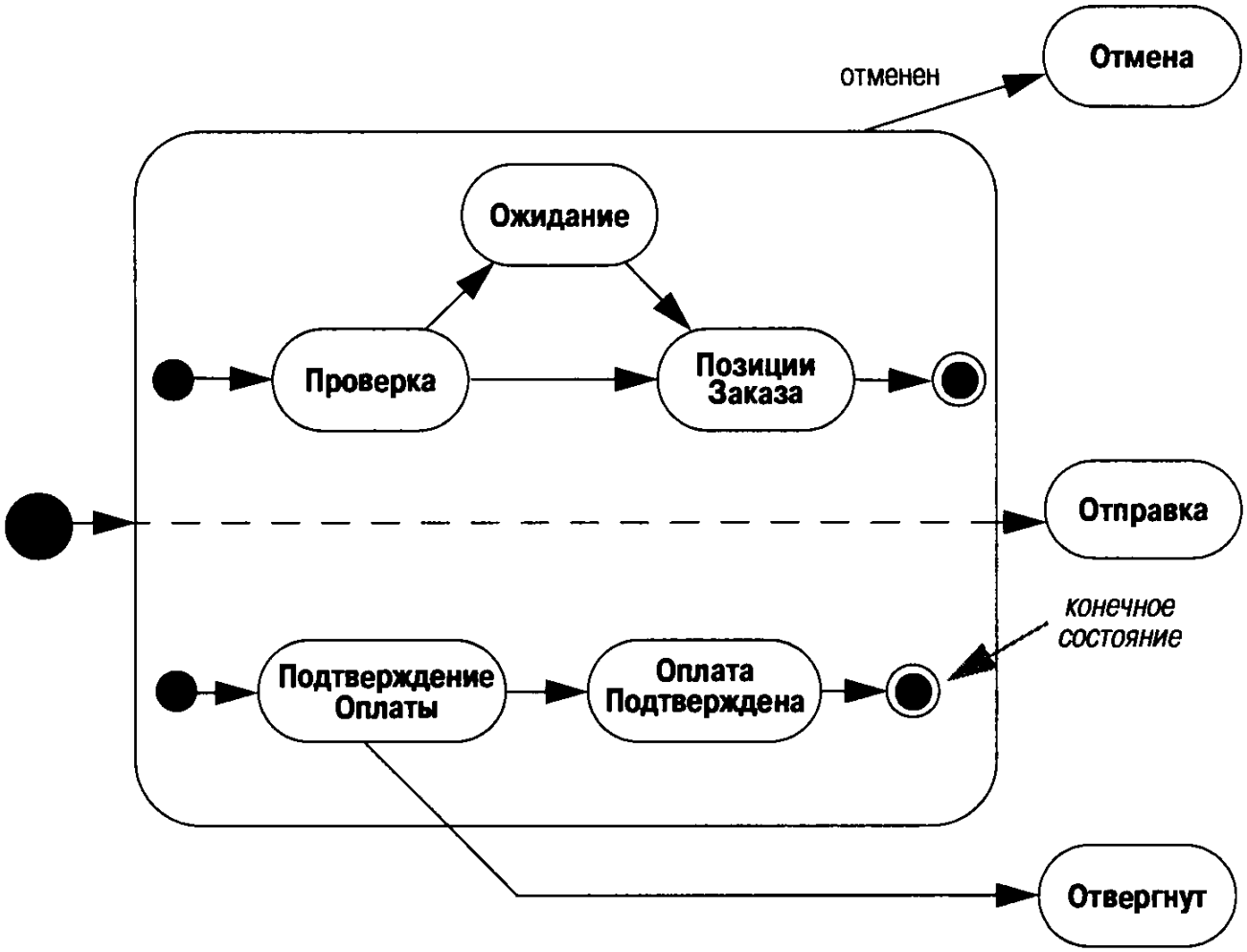


Рис. 8.5. Диаграмма параллельных состояний

Смысл параллельных секций диаграммы состояний заключается в том, что в любой момент времени данный заказ находится одновременно в двух различных состояниях, каждое из которых относится к своей исходной диаграмме. Когда заказ покидает параллельные состояния, он оказывается только в одном состоянии. Из этой диаграммы можно увидеть, что в начальный момент заказ оказывается одновременно в двух состояниях: Проверка Позиции Заказа и Подтверждение Оплаты. Если деятельность «проверить оплату» в состоянии Подтверждение Оплаты успешно завершится первой, то заказ окажется в двух состояниях: Проверка Позиции Заказа и Оплата Подтверждена. Если же наступит событие «отменен», то заказ окажется только в состоянии Отмена.

Диаграммы параллельных состояний полезны в тех ситуациях, когда некоторый объект обладает множеством независимых поведений. Отметим, однако, что не следует создавать слишком большое количество параллельных состояний, описывающих поведение одного объекта. Если для некоторого объекта имеется несколько достаточно сложных диаграмм параллельных состояний, то следует рассмотреть возможность разделения этого объекта на отдельные объекты.

Когда использовать диаграммы состояний

Диаграммы состояний являются хорошим средством для описания поведения некоторого объекта в нескольких различных вариантах использования. Однако они не слишком пригодны для описания поведения нескольких объектов, образующих кооперацию. По существу, диаграммы состояний полезно объединять с другими методами.

Например, диаграммы взаимодействия (см. главу 5) являются хорошим средством для описания поведения нескольких объектов в одном варианте использования, а диаграммы деятельности (см. главу 9) удобны для представления общей последовательности действий для нескольких объектов и вариантов использования.

Не все разработчики считают диаграммы состояний естественными. Следует внимательно присмотреться к тому, как аналитики работают с этими диаграммами. Может оказаться, что ваша команда считает диаграммы состояний бесполезными, потому что они не вписываются в ее стиль работы. Это не столь большая проблема; как всегда, нужно помнить, что следует пользоваться именно теми средствами, которые больше подходят для вашей работы.

Если вы все-таки пользуетесь диаграммами состояний, не пытайтесь строить их для каждого класса в вашей системе. Хотя строгие формалисты зачастую применяют именно такой подход, это почти всегда будет пустой тратой времени. Используйте диаграммы состояний только для тех классов, поведение которых вас действительно интересует, и если построение диаграммы состояний помогает лучше его понять. Многие разработчики считают, что пользовательский интерфейс и управляющие объекты обладают именно таким поведением, которое полезно изображать с помощью диаграмм состояний.

Где найти дополнительную информацию

Как в «Руководстве пользователя» (Буч, Рамбо и Джекобсон, 1999 [6]), так и в «Справочнике пользователя» (Рамбо, Джекобсон и Буч, 1999 [37]) можно найти дополнительную информацию по диаграммам состояний. Многие разработчики-практики склонны довольно часто использовать модели состояний, поэтому неудивительно, что в книге Дугласа (Douglass), 1998 [17] много говорится о диаграммах состояний, включая различные аспекты их реализации.

Если вы планируете интенсивно использовать диаграммы, то следует обратиться к книге Кука и Дэниелса (1994) [13]. Хотя имеются различия в семантике обозначений схем состояний и диаграмм состояний в языке UML, авторы подробно рассматривают такие вопросы, в которых желательно разбираться, если вы пользуетесь диаграммами состояний.

9

Диаграммы деятельности

Диаграммы деятельности – это одна из самых больших неожиданностей языка UML.

В отличие от большинства других конструкций языка UML диаграммы деятельности не имеют явно выраженного источника в предыдущих работах «троих друзей». Напротив, **диаграмма деятельности** соединяет в себе идеи нескольких различных методов: диаграмм событий Джима Оделла, методов моделирования состояний SDL, моделирования потоков работ и сетей Петри. Эти диаграммы особенно полезны в сочетании с потоками работ, а также при описании поведения, включающего в себя большое количество параллельных процессов.

Я постараюсь рассмотреть диаграммы деятельности более подробно, чем это действительно необходимо для столь небольшой книги. Причиной может служить то обстоятельство, что диаграммы деятельности являются одной из наименее понятных областей языка UML, а в изданных книгах по UML эта тема представлена недостаточно полно.

На рис. 9.1 основным элементом является **состояние деятельности** или просто **деятельность.** Деятельность представляет собой некоторое состояние, в котором что-либо выполняется: будь то процесс реального времени, такой как написание письма, либо исполнение компьютерной программы, такой как метод некоторого класса.

Диаграмма деятельности описывает последовательность подобных деятельностей, позволяя при этом одновременно изображать как условное, так и параллельное поведение. Диаграмма деятельности по сути

представляет собой вариант диаграммы состояний, в которой большинство, а может быть и все состояния являются состояниями деятельности. Таким образом, большая часть терминологии совпадает с терминологией диаграмм состояний.

Условное поведение изображается с помощью ветвлений и соединений.

Ветвление имеет единственный входящий переход и несколько выходящих переходов со сторожевыми условиями. Поскольку может выполняться только один из выходящих переходов, сторожевые условия должны взаимно исключать друг друга. Если в качестве сторожевого условия используется [иначе], то это означает, что переход с меткой «иначе» должен произойти в том случае, когда все другие сторожевые условия для данного ветвления являются ложными.

На рис. 9.1 после заполнения бланка заказа имеется ветвление. Если заказ оказывается срочным, выполняется его срочная доставка, в противном случае – обычная доставка.

Соединение имеет несколько входящих переходов и единственный выходящий переход. Соединение означает окончание условного поведения, которое было начато соответствующим ветвлением.

Ветвления и соединения можно указывать явным образом с помощью ромба. Состояние деятельности, так же как и любое другое состояние, может иметь несколько выходящих переходов со сторожевыми условиями и несколько входящих переходов. Чтобы сделать ветвления и соединения более понятными на диаграмме, следует использовать ромбы.

Параллельное поведение изображается с помощью слияний и разделений. **Разделение** имеет единственный входящий переход и несколько выходящих переходов. Когда срабатывает входящий переход, все выходящие переходы выполняются параллельно. Таким образом, после поступления заказа заполнение бланка заказа и выставление счета выполняются параллельно (рис. 9.1).

Данная диаграмма утверждает, что эти деятельности могут осуществляться параллельно. По существу, это означает, что последовательность их выполнения может быть произвольной. Можно вначале заполнить бланк заказа, выставить счет и отправить товар, после чего получить оплату. Или можно вначале выставить счет, получить оплату, после чего заполнить бланк заказа и отправить товар. Все эти варианты допускаются рассматриваемой диаграммой.

Эти деятельности можно сделать чередующимися. Так, например, взять первую позицию заказа со склада, напечатать счет, затем вторую позицию заказа, поместить счет в конверт и т. д. Или выполнять некоторые из этих деятельностей одновременно: печатать счет и проверять наличие товаров на складе. Согласно нашей диаграмме, любой из этих вариантов является корректным.

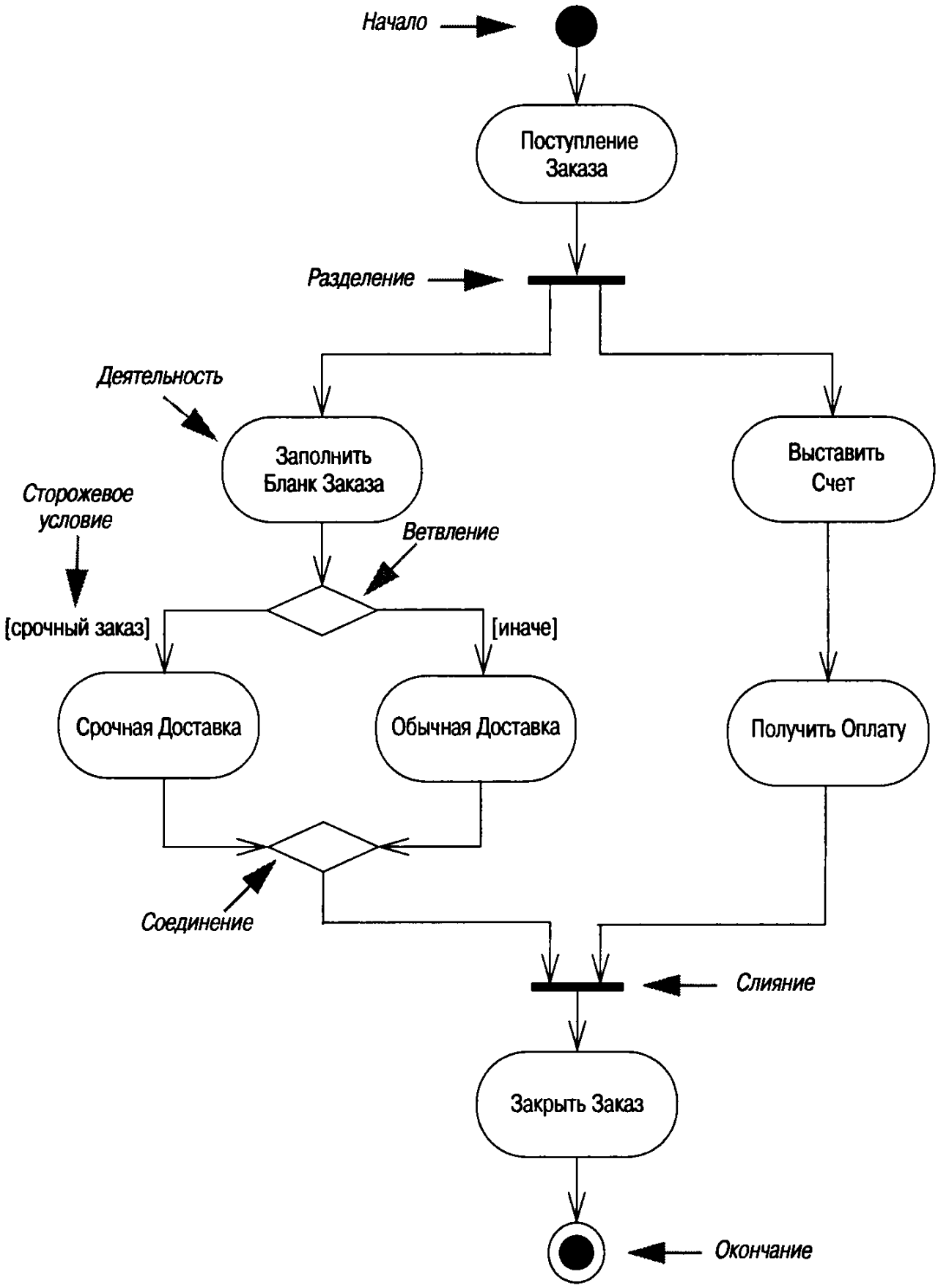


Рис. 9.1. Диаграмма деятельности

Рассматриваемая диаграмма деятельности позволяет выбрать отдельный заказ, с которым необходимо что-либо сделать. Другими словами, данная диаграмма просто устанавливает основные правила последовательности действий, которые необходимо соблюдать. В этом заключает-

ся главное различие между диаграммой деятельности и схемой потоков: схемы потоков обычно ограничены последовательными процессами, в то время как диаграммы деятельности могут описывать параллельные процессы.

Это необходимо для моделирования бизнес-систем, т. к. они зачастую имеют не только последовательные бизнес-процессы. В подобных случаях становится чрезвычайно полезным метод, позволяющий поддерживать параллельное поведение. Такой метод особенно нужен тем аналитикам, которые желают моделировать поведение, не являющееся безусловно последовательным, и использовать возможности данного метода для распараллеливания отдельных операций. Все это может повысить эффективность и улучшить ответственность бизнес-процессов.

Диаграммы деятельности оказываются весьма полезными для параллельных программ, поскольку они позволяют изобразить графически нити процесса и их синхронизацию по времени исполнения.

В процессе моделирования некоторого параллельного поведения необходима синхронизация. Мы не можем закрыть заказ до тех пор, пока он не будет оплачен и отправлен клиенту. Именно это указывает **слияние** перед деятельностью Закрыть Заказ. Слияние на диаграмме деятельности означает, что выходные переходы могут произойти только в том случае, когда состояния у всех входящих переходов завершат свои деятельности.

Разделения и слияния должны соответствовать друг другу. В простейшем случае это означает, что для любого разделения на диаграмме должно иметься соответствующее слияние, которое объединяет все нити, имеющие начало в этом разделении. (Это правило обусловлено тем обстоятельством, что диаграмма деятельности является, по существу, разновидностью диаграммы состояний.)

Однако это правило имеет несколько исключений:

- Нить, выходящая из некоторого разделения, сама может быть разделением с новыми нитями, которые объединяются вместе до того, как будет достигнуто слияние всех исходных нитей.
- Если выходящая из некоторого разделения нить сразу попадает в другое разделение, то это второе разделение можно удалить, а выходящие из него нити изобразить выходящими из первого разделения. Таким образом на рис. 9.2 удалено разделение между деятельностями по приготовлению еды и исходным разделением. Аналогично, если некоторое слияние непосредственно переходит в другое слияние, то первое слияние можно удалить, а все входящие в него нити изобразить входящими во второе слияние. Это упрощение нотации позволяет преодолеть ненужное усложнение диаграмм, и точно такая же семантика позволяет изображать на диаграмме дополнительные разделения и слияния.

- Для синхронизации нитей может использоваться дополнительная конструкция, которая называется **синхронизирующим состоянием**. Это позволяет избежать путаницы при интерпретации правил разделения и слияния для нескольких параллельных нитей. Более подробно с этой конструкцией можно познакомиться, обратившись к «Справочнику пользователя» (Рамбо, Джекобсон и Буч, 1999 [37]) или стандартной документации.

Данное правило имеет еще одно исключение, при котором все входные состояния некоторого слияния должны быть завершены до того, как это слияние сможет произойти. Вы можете добавить дополнительное условие на отдельную нить, выходящую из разделения. Результатом этого является так называемая **условная нить**. Если в ходе выполнения процесса условие такой условной нити принимает значение ложь, это означает, что данная нить должна быть завершена для выполнения последующего слиянии. Так, из диаграммы на рис. 9.2 следует, что даже если у меня нет желания пить вино, все же я хотел бы иметь возможность съесть спагетти с соусом Карбонара. (Должен сознаться, что, к сожалению, мне так и не удалось проверить это правило в процессе работы над данной диаграммой.)

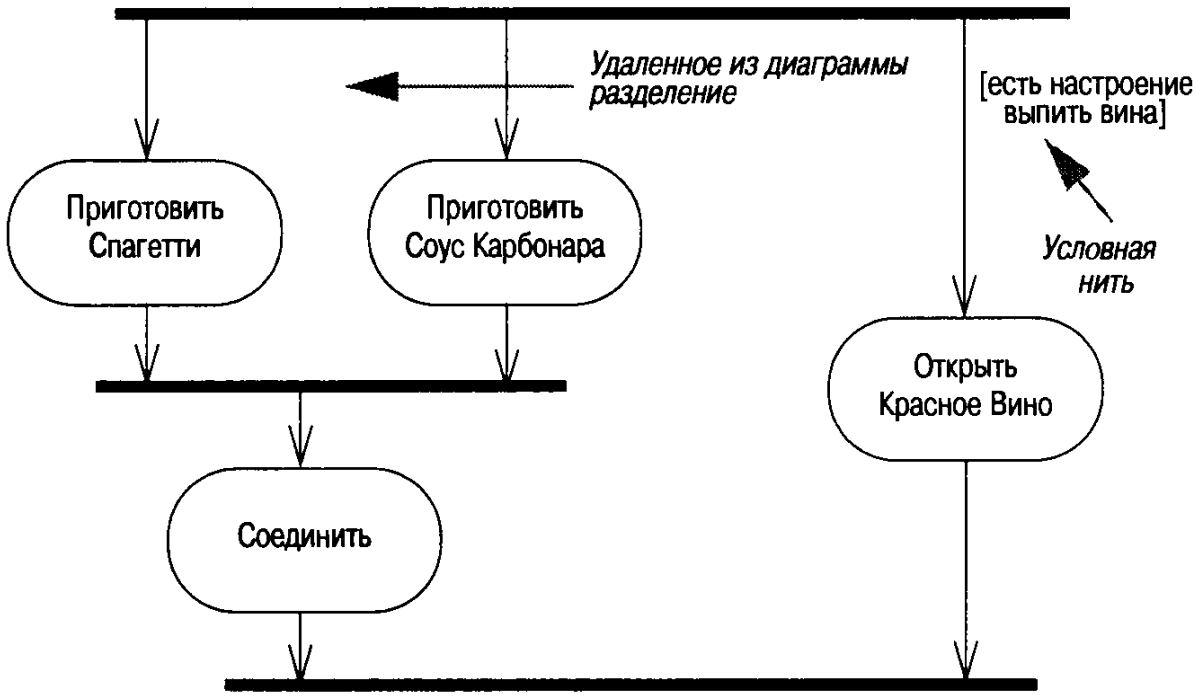


Рис. 9.2. Разделения, слияния и условные потоки

Декомпозиция деятельности

Деятельность может быть разделена на поддеятельности. Ситуация здесь во многом аналогична изображению суперсостояния и подсостояний на диаграмме состояний. Можно показать только суперсостояние на родительской (порождающей) диаграмме или показать суперсостояние и относящееся к нему поведение внутри этого суперсостоя-

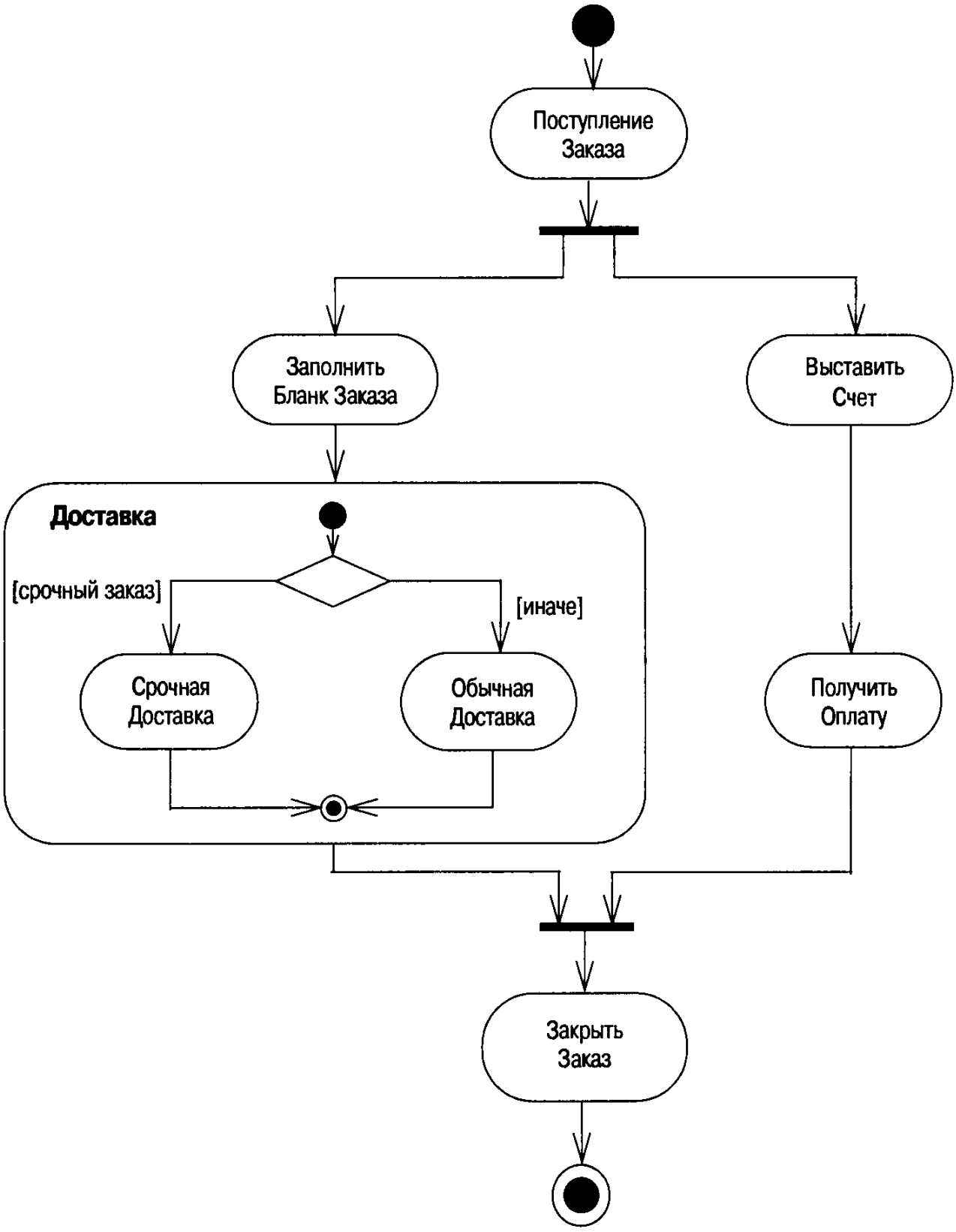


Рис. 9.3. Использование составной деятельности по доставке

ния, как это представлено на рис. 9.3. В данном случае деятельность по доставке выделена как отдельная со своим началом и окончанием. Можно также изобразить переходы, входящие или выходящие непосредственно из дочерней диаграммы. Достоинство явного указания начального и конечного состояний состоит в том, что в этом случае деятельность по доставке может быть использована в нескольких кон-

текстах, и родительская диаграмма не будет зависеть от содержимого дочерней диаграммы.

Динамическая параллельность

Динамическая параллельность позволяет представлять итерации без изображения таких конструкций, как петля.

На рис. 9.4 деятельность Заполнить Строку Заказа выполняется только один раз для каждой позиции заказа. Маркер кратности (*) указывает, что эта деятельность может выполняться несколько раз. Переход к деятельности Доставка Заказа происходит только в том случае, когда все позиции отдельного заказа будут заполнены. Если несколько деятельностей должны некоторым образом выполняться совместно, это можно показать, пометив деятельность Заполнить Строку Заказа как составную.

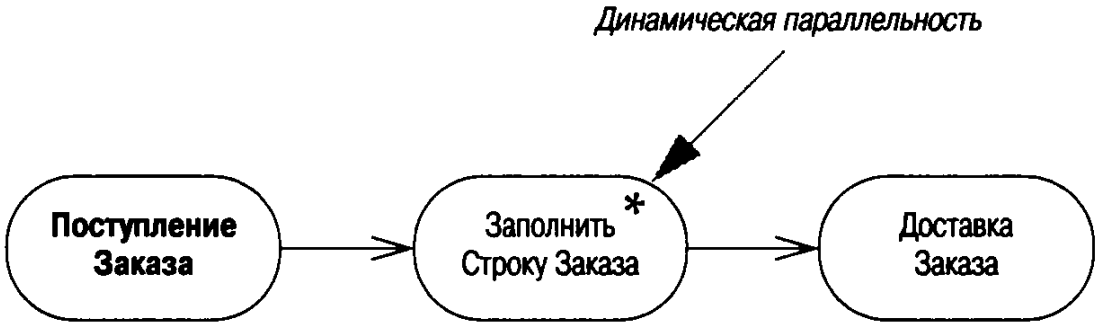


Рис. 9.4. Динамическая параллельность

Дорожки

Диаграммы деятельности говорят нам о том, что происходит, однако они ничего не говорят о том, кто участвует в выполнении соответствующих деятельностей. С точки зрения программирования это означает, что данная диаграмма не сообщает, какой класс отвечает за выполнение той или иной деятельности.

При моделировании предметной области это означает, что диаграмма не может сообщить, какие люди или подразделения отвечают за выполнение каждой деятельности. Один из способов решить эту проблему состоит в том, чтобы снабдить каждую деятельность меткой класса или человека, ответственного за ее выполнение. Это дает нужный результат, однако данный способ не обеспечивает такой ясности, как в диаграммах взаимодействия (см. главу 5) при изображении обмена сообщениями между объектами.

Другой способ решить проблему – изобразить так называемые **дорожки** (swimlanes).

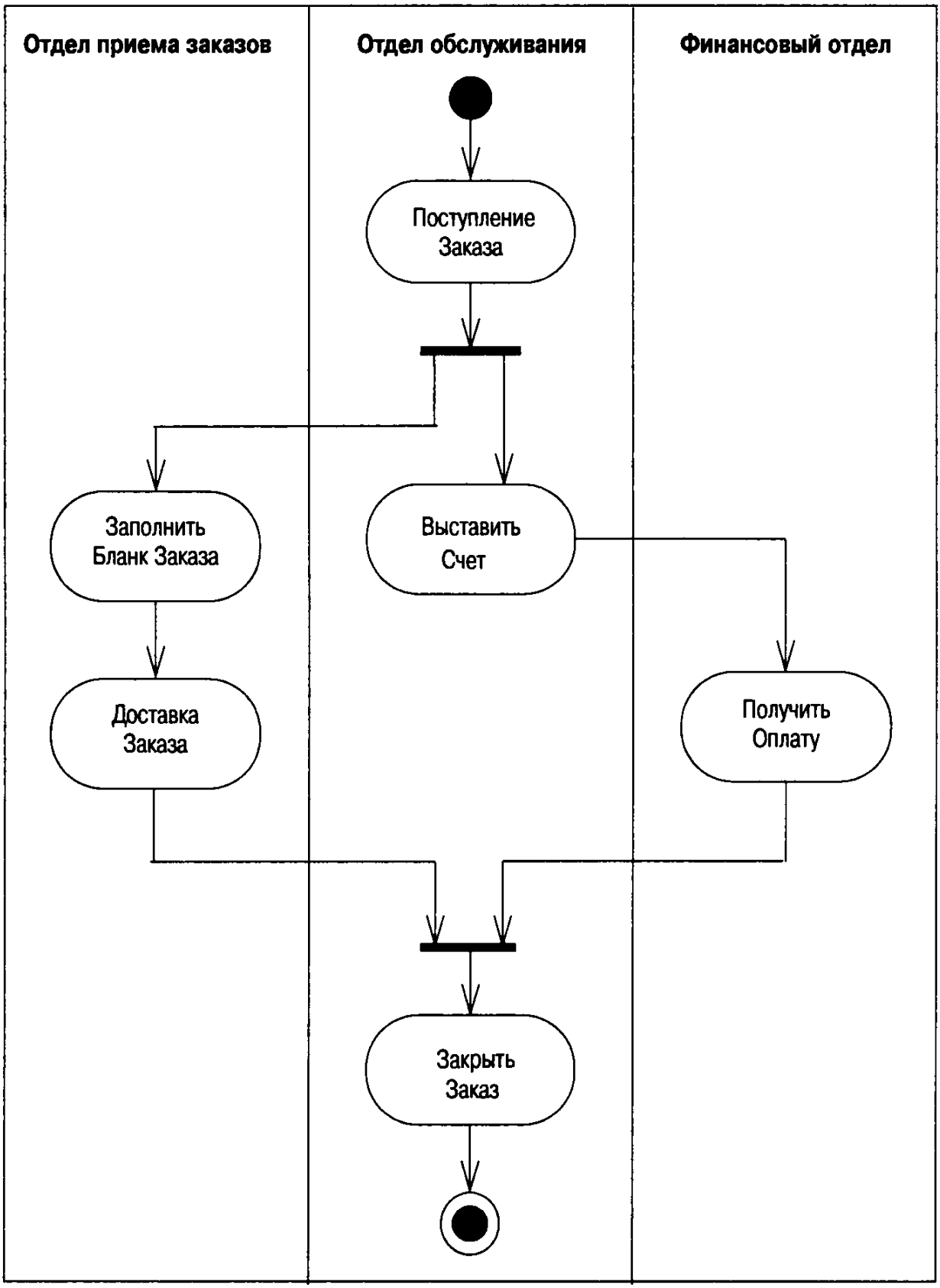


Рис. 9.5. Дорожки

Для использования дорожек необходимо с помощью пунктирных линий разделить диаграммы деятельности на вертикальные зоны. Каждая зона представляет собой зону ответственности конкретного класса или отдельного организационного подразделения, как это изображено на рис. 9.5.

Преимущество дорожек проявляется в том, что они позволяют объединить логику представления диаграммы деятельности с представлением ответственности на диаграмме взаимодействия. Однако построение комплексной диаграммы может вызвать затруднения. Я использовал даже нелинейные зоны в тех ситуациях, когда это является лучшим, чем ничего. (Иногда следует вообще прекратить попытки представить слишком много информации на одной диаграмме.)

Некоторые разработчики при построении диаграмм деятельностей стараются сразу назначить объектам ответственность за деятельности. Другим нравится начинать работу с построения исходных диаграмм деятельности, поскольку эти диаграммы дают им общее представление о поведении системы, а назначение деятельностям соответствующих объектов выполняется позднее. Мне приходилось встречать разработчиков, которые выполняли такое назначение немедленно и весьма эмоционально высказывались в адрес тех, кто откладывал это на потом; эти разработчики обвиняли своих коллег в том, что они занимаются построением обычных диаграмм потоков данных, а вовсе не объектно-ориентированным проектированием.

Должен признаться, что я сам иногда изображаю диаграмму деятельности без назначения поведения конкретным объектам, откладывая это в долгий ящик. Я считаю, что для пользы дела лучше в тот или иной момент времени заниматься решением только одной проблемы. Это особенно справедливо при моделировании бизнес-процессов, тем самым эксперту предметной области предоставляется возможность обрести новый взгляд на вещи. Этот способ работает на меня. Другие предпочитают сразу же назначать поведение объектам. Вы же поступайте так, как считаете удобным для себя. Важно помнить, что деятельности нужно назначить классам до окончания моделирования. Что касается меня, то я предпочитаю для этой цели использовать диаграммы взаимодействия (см. главу 5).

Когда использовать диаграммы деятельности

Подобно большинству других методов моделирования диаграммы деятельности обладают определенными достоинствами и недостатками, поэтому их лучше всего использовать в сочетании с другими методами.

Самым большим достоинством диаграмм деятельности является возможность представления и описания параллельного поведения. Именно благодаря этому они являются мощным средством моделирования потоков работ и, по существу, программирования многопоточности. Самый большой недостаток этих диаграмм заключается в том, что они не позволяют представить связи между действиями и объектами достаточно наглядно.

Связь с некоторым объектом можно определить, помечая ту или иную деятельность именем соответствующего объекта или используя дорожки, которые разделяют диаграмму деятельности на зоны ответственности, но этот способ не обладает наглядностью диаграмм взаимодействия (см. главу 5). По этой причине некоторые разработчики считают, что использование диаграмм деятельности не отвечает объектно-ориентированному подходу и поэтому их вовсе не стоит строить. Я же считаю этот метод весьма полезным и не собираюсь от него отказываться.

Я предпочитаю использовать диаграммы деятельности в следующих ситуациях:

- *Анализ варианта использования.* На этом этапе меня не интересует связь между действиями и объектами; мне только нужно понять, какие действия должны иметь место и каковы зависимости в поведении системы. Я выполняю связывание методов с объектами позже и показываю эти связи с помощью диаграмм взаимодействия.
- *Понимание потока работ.* Прежде чем приступить к рассмотрению содержания вариантов использования, целесообразно привлечь диаграммы деятельности для лучшего понимания соответствующего бизнес-процесса. Эти диаграммы лучше разрабатывать совместно с бизнес-аналитиками, поскольку при этом можно понять особенности бизнес-процесса и возможности его изменения.
- *Описание сложного последовательного алгоритма.* В этом случае диаграмма деятельности не позволяет представить ничего сверх того, что может быть изображено на согласованной с обозначениями языка UML схеме алгоритма. При этом можно использовать принятые на схемах алгоритмов специальные обозначения.
- *Работа с многопоточными приложениями.* Я сам еще не использовал диаграммы деятельности для этой цели, однако слышал положительные отзывы о подобном применении.

Диаграммы деятельности не следует использовать в следующих ситуациях:

- *Пытаться представить кооперацию объектов.* Диаграммы взаимодействия являются более простыми и обеспечивают более наглядное представление кооперации.
- *Пытаться представить поведение объектов в течение их жизненного цикла.* Для этой цели лучше использовать диаграмму состояний (см. главу 8).
- *Представление сложных логических условий.* Для этой цели лучше использовать таблицу истинности.

В ходе работы над версией 1.3 языка UML диаграммы деятельности были значительно уточнены и существенно дополнены. Однако все это вызывает смешанные чувства. Проблема состоит в том, что если ис-

пользовать эти диаграммы для концептуального моделирования, многие из уточнений теряют смысл. В подобных ситуациях не следует стремиться к исчерпывающей точности – нужно лишь представить общую картину того, как все работает. Даже если такие уточнения очевидны, все равно едва ли стоит что-либо исправлять, пока вы не будете способны проверить и протестировать диаграмму. Следует помнить фразу Бертрана Мейера: «Пеной не напьешься».

С другой стороны, наличие некоторого стандарта для представления диаграмм состояний и диаграмм деятельности дает более стабильную основу для разработки инструментальных средств, применение которых позволило бы реализовывать эти диаграммы. Именно такие средства позволят вам разрабатывать данные диаграммы и тестировать их.

Где найти дополнительную информацию

По диаграммам деятельности имеется довольно мало информации. В «Справочнике пользователя» (Рамбо, Джекобсон и Буч, 1999 [37]) рассматривается много деталей, но совсем не объясняется, как они работают. «Руководство пользователя» (Буч, Джекобсон и Рамбо, 1999 [6]) вовсе не содержит подробные ответы на типичные вопросы, возникающие при попытке использовать эти диаграммы. Было бы неплохо, если бы кто-нибудь заполнил этот пробел в ближайшее время.

10

Физические диаграммы

В языке UML имеется два вида физических диаграмм: диаграммы развертывания и диаграммы компонентов.

Диаграммы развертывания

Диаграмма развертывания (deployment diagram) отражает физические взаимосвязи между программными и аппаратными компонентами разрабатываемой системы. Эта диаграмма является хорошим средством для представления маршрутов перемещения объектов и компонентов в распределенной системе.

Каждый **узел** на диаграмме развертывания представляет собой некоторый тип вычислительного устройства – в большинстве случаев самостоятельную часть аппаратуры. Эта аппаратура может быть как простым устройством или датчиком, так и мэйнфреймом.

На рис. 10.1 изображен персональный компьютер, соединенный с Unix-сервером посредством протокола TCP/IP. **Соединения** между узлами показывают физические каналы связи, с помощью которых осуществляются взаимодействия в системе.

Диаграммы компонентов

Диаграмма компонентов (component diagram) показывает различные компоненты системы и зависимости между ними.

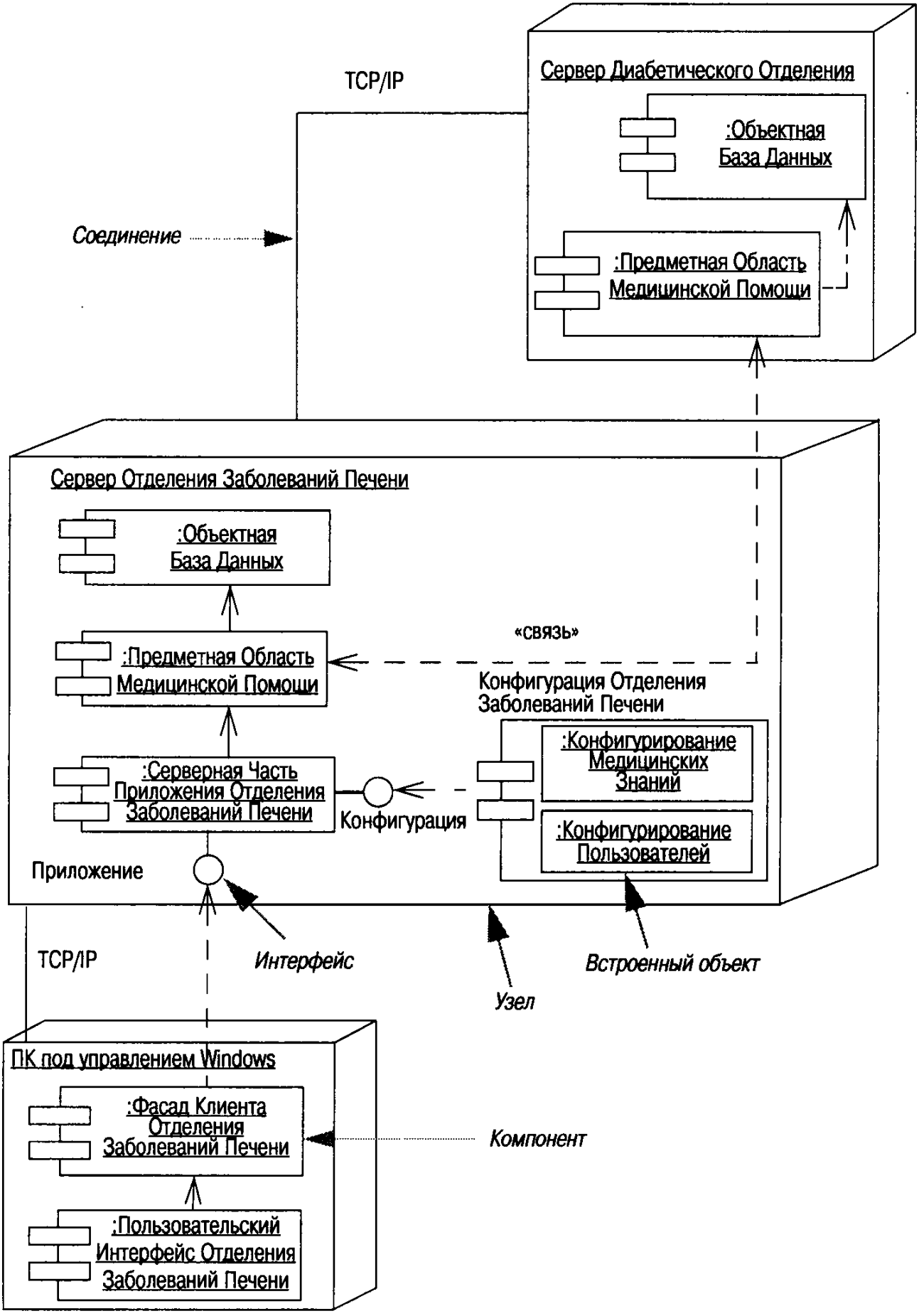


Рис. 10.1. Диаграмма развертывания

Компонент представляет собой физический модуль программного кода. Компонент часто считают синонимом пакета, но эти понятия могут отличаться, поскольку компоненты представляют собой физическое объединение программного кода. Хотя отдельный класс может быть представлен в целой совокупности компонентов, этот класс должен быть определен только в одном пакете. Например, класс *Строка* в языке Java является частью пакета java.lang, но он может быть обнаружен в ряде компонентов.

Зависимости между компонентами показывают, как изменения одного компонента могут повлиять на изменения других компонентов. Существует довольно ограниченное количество видов зависимостей, которые можно использовать, включая зависимости типа связь и компиляция. Я часто применяю эти виды зависимостей для представления связей компонентов между собой.

Объединение диаграмм компонентов и развертывания

Хотя диаграммы развертывания и диаграммы компонентов можно изображать отдельно, также допускается помещать диаграмму компонентов на диаграмму развертывания, как показано на рис. 10.1. Это целесообразно делать, чтобы показать какие компоненты выполняются и на каких узлах.

Так, например, на данной диаграмме компоненты Пользовательский Интерфейс Отделения Заболеваний Печени и Фасад Клиента Отделения Заболеваний Печени исполняются на ПК под управлением ОС Windows. Компонент Пользовательский Интерфейс Отделения Заболеваний Печени зависит от компонента Фасад Клиента Отделения Заболеваний Печени, поскольку он обращается к конкретным методам этого Фасада. Хотя связь является двунаправленной в том смысле, что Фасад возвращает данные, компонент Фасад не знает, кто его вызывает, и поэтому не зависит от компонента Пользовательский Интерфейс. С другой стороны, связь между двумя компонентами Предметная Область Медицинской Помощи является двунаправленной, поскольку каждый из них знает, какому компоненту он передает данные. Однако эти компоненты выполняются на отдельных узлах.

Компонент может иметь более одного интерфейса, при этом в каждом случае видно, какие компоненты взаимодействуют с тем или иным интерфейсом. На рис. 10.1 серверная часть приложения имеет два интерфейса. Один интерфейс используется фасадом приложения во время его исполнения на ПК, другой интерфейс используется компонентом конфигурирования во время его исполнения на сервере.

Факт использования нескольких компонентов Предметная Область Медицинской Помощи скрыт от своих клиентов. Каждый компонент

Предметная Область Медицинской Помощи имеет свою локальную базу данных.

Разработчики часто изображают на физических диаграммах специальные символы, которые по своему внешнему виду напоминают различные элементы. Например, применяются специальные пиктограммы для серверов, ПК и баз данных. В рамках языка UML это вполне допускается: каждую из пиктограмм можно рассматривать как стереотип соответствующего элемента диаграммы. Обычно такие пиктограммы способствуют лучшему пониманию диаграммы, хотя и усложняют ее, если одновременно изображаются узлы и компоненты, как на рис. 10.1.

Когда использовать физические диаграммы

Многие разработчики творчески подходят к использованию этого вида информации, однако другие строят эти диаграммы формально с целью соответствия стандартам языка UML. Сам я разрабатываю эти диаграммы всякий раз, когда хочу представить физическую информацию, которая отличается от соответствующей логической информации. В большинстве случаев я рисую по меньшей мере одну диаграмму, на которой показаны узлы и основные компоненты в стиле рис. 10.1. При этом я стараюсь использовать графические пиктограммы, если это не приводит к излишнему усложнению диаграммы.

11

Язык UML и программирование

Итак, мы практически полностью рассмотрели нотацию языка UML. Остался один большой вопрос: как обычный программист может реально использовать язык UML в повседневной однообразной работе? Попытаюсь ответить на этот вопрос, повествуя о том, как я сам использую язык UML при программировании, хотя и в небольшом масштабе. Я не буду слишком вдаваться в детали, но надеюсь, это поможет вам понять, что можно делать с помощью языка UML.

Представим себе компьютерную систему, которая проектируется для сбора информации о пациентах больницы.

Различные специалисты-медики наблюдают своих пациентов. Эта несложная система позволит каждому из них получать информацию о проведенных наблюдениях и вводить новую информацию. Поскольку объем данной книги ограничен, я не буду касаться базы данных и пользовательского интерфейса, а рассмотрю только основные классы предметной области.

Пример настолько прост, что может быть описан единственным вариантом использования под названием «просмотр и ввод данных наблюдения пациентов». Мы можем конкретизировать его с помощью следующих сценариев:

- Запросить последние данные сердечного ритма пациента.
- Запросить группу крови пациента.
- Обновить данные об уровне сознания пациента.

- Обновить данные о сердечном ритме пациента. Система определяет ритм как медленный, нормальный или быстрый в соответствии с заданными в системе диапазонами.

Мой первый шаг в этом процессе состоит в разработке концептуальной модели, которая описывает понятия данной предметной области. На этой стадии я не задумываюсь, как будет реализовано соответствующее программное обеспечение, а думаю только о том, как выстроить систему понятий в представлении докторов и медсестер. Я начну с модели, основанной на нескольких образцах анализа из моей книги (Фаулер, 1997 [18]): «Наблюдение» (Observation), «Количество» (Quantity), «Диапазон» (Range) и «Показатель с диапазоном» (Phenomenon with Range).

Наблюдение пациента: модель предметной области

На рис. 11.1 показана начальная модель предметной области для нашей системы.

Каким образом эти понятия представляют информацию о данной предметной области?

Я начну с самых простых понятий: Количество, Единица и Диапазон. Количество представляет собой значение, обладающее размерностью, например 6 футов – количество, величина которого равна 6, а единица измерения – фут. Единицы просто представляют собой те категории измерения, которые нужны нам для работы. Диапазон позволяет рассматривать их как единичное понятие, например, диапазон от 4 до 6 футов представляется как единственный объект «Диапазон с верхней границей 6 футов и нижней границей 4 фута». В общем случае диапазоны могут быть выражены в терминах, допускающих сравнение (с использованием операторов <, >, <=, >= и =), таким образом, и верхняя и нижняя границы некоторого Диапазона являются некоторыми величинами (Количество представляет собой разновидность величины).

Каждое выполненное доктором или медсестрой наблюдение представляет собой экземпляр понятия Наблюдение и является либо Измерением, либо Категорией Наблюдения. Таким образом, измерение роста в 6 футов для Мартина Фаулера следует представить как экземпляр Измерения. С этим Измерением ассоциированы величина «6 футов», Тип Показателя «рост» и Пациент по имени Мартин Фаулер. Типы Показателей представляют собой измеримые величины: рост, вес, сердечный ритм и т. д.

Некоторое наблюдение, согласно которому Мартин Фаулер имеет группу крови О, следует представить как Категорию Наблюдения, с которой

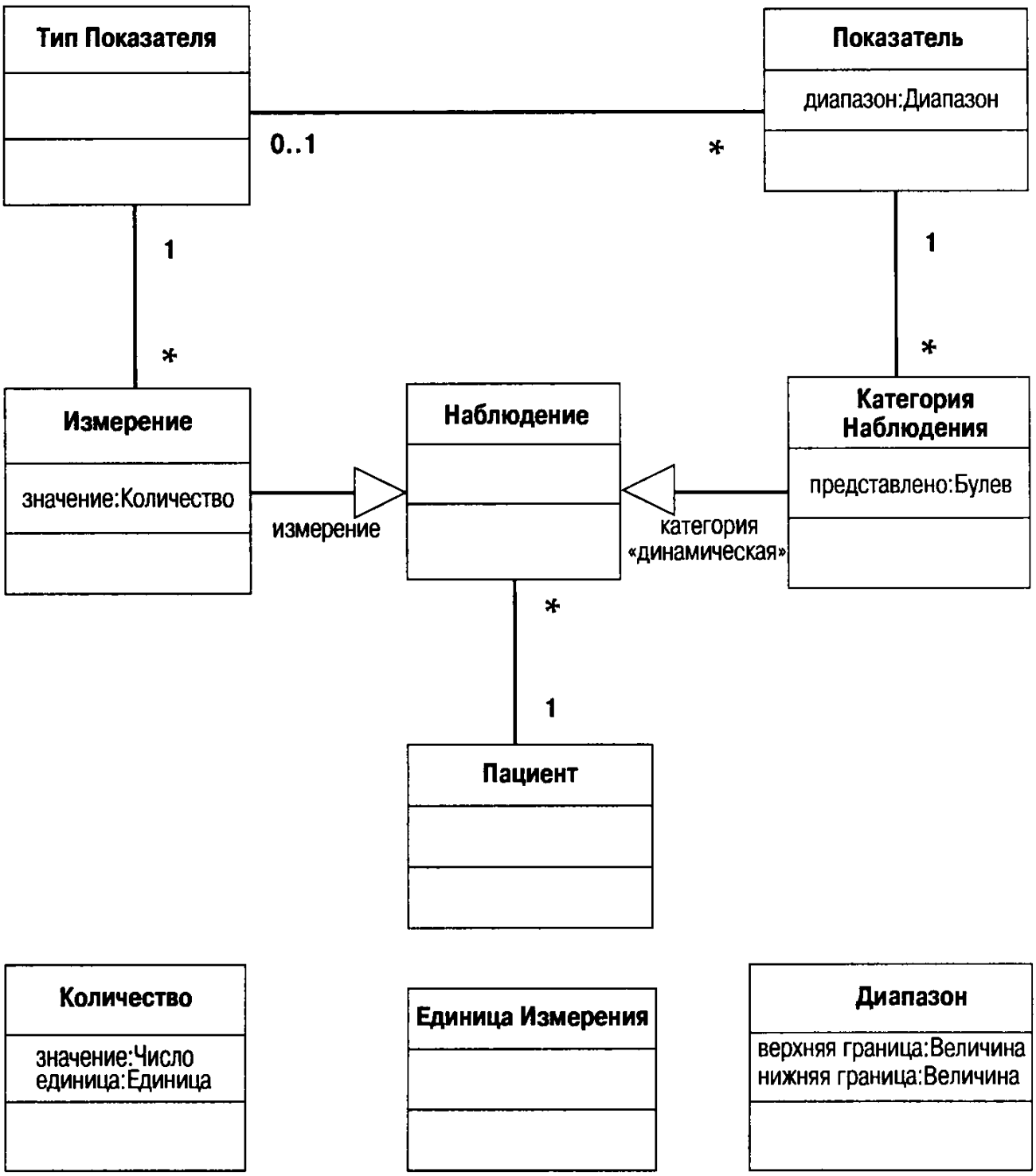


Рис. 11.1. Модель предметной области наблюдения пациента

ассоциирован Показатель «группа крови О». Этот Показатель в свою очередь связан с Типом Показателя «группа крови».

Диаграмма объектов на рис. 11.2 может несколько прояснить данную ситуацию.

Рис. 11.3 иллюстрирует, что можно выполнить Наблюдение, которое служит одновременно Измерением и Категорией Наблюдения. Основой этого факта служит то, что Измерение «90 ударов в минуту» может также являться Категорией Наблюдения, с которой связан Показатель «быстрый сердечный ритм».

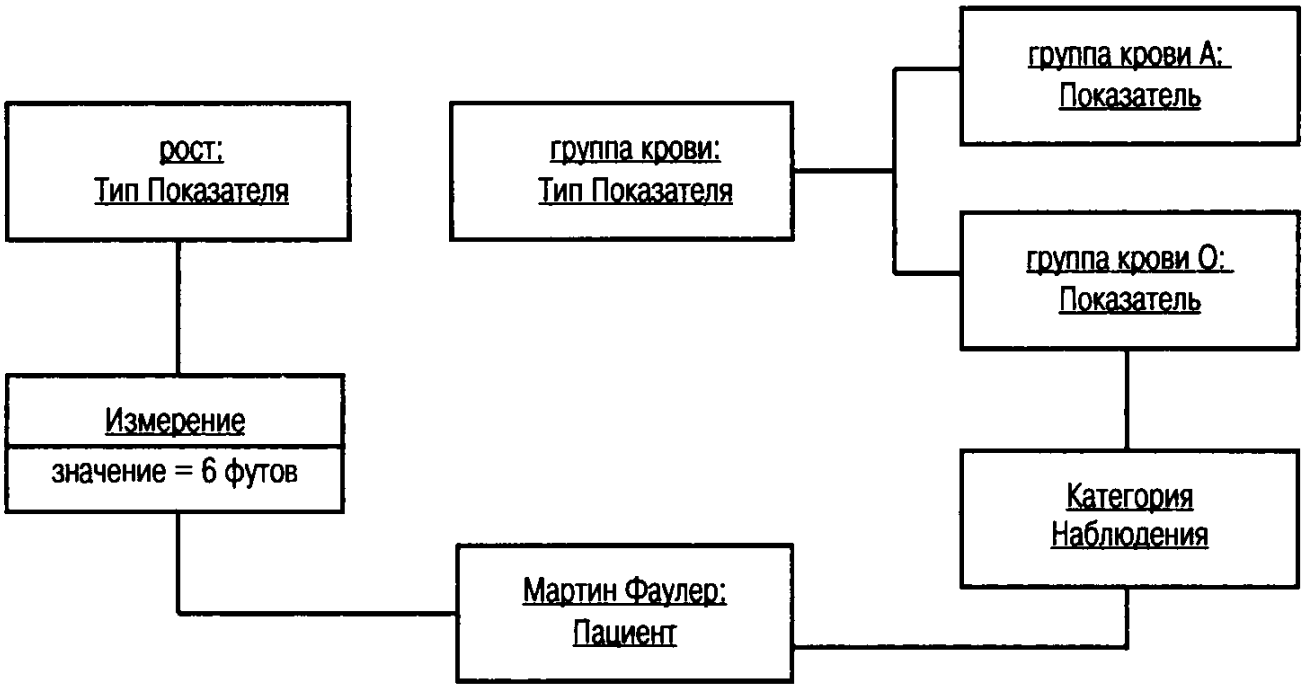


Рис. 11.2. Диаграмма объектов наблюдения пациента

На данной стадии мною рассмотрены только сами понятия без учета их поведения. Я не всегда поступаю именно так, однако данное представление оказывается подходящей отправной точкой для решения задачи, связанной главным образом с информацией.

Мною до сих пор рассматриваются *понятия*, связанные с наблюдением пациентов, как если бы я имел дело с доктором или медсестрой. (На самом деле, все так и есть. Концептуальные модели были построены с моей помощью парой докторов и медсестрой.) Чтобы перейти к объектно-ориентированному программированию, необходимо решить, как рассматривать это концептуальное представление в терминах программного обеспечения. (Должен же я как-нибудь вставить в эту книгу код на Java!)

Большинство рассмотренных понятий могут быть преобразованы в классы языка Java. Понятия Пациент, Тип Показателя, Показатель, Единица Измерения и Количество преобразуются без проблем. Проблема возникает только с понятиями Диапазон и Наблюдение.

Проблема с Диапазоном обусловлена тем, что мне нужно сформировать количественный диапазон для Показателя. Это можно было бы осуществить, создав интерфейс «величина» и установив, что Количество реализует данный интерфейс, но это привело бы к определенным трудностям. В языке Smalltalk подобных проблем не возникает, да и в языке C++ для этой цели можно воспользоваться параметризованными типами. Что касается данного примера, то здесь целесообразно использовать класс КоличественныйДиапазон, который, в свою очередь, использует образец «Диапазон».

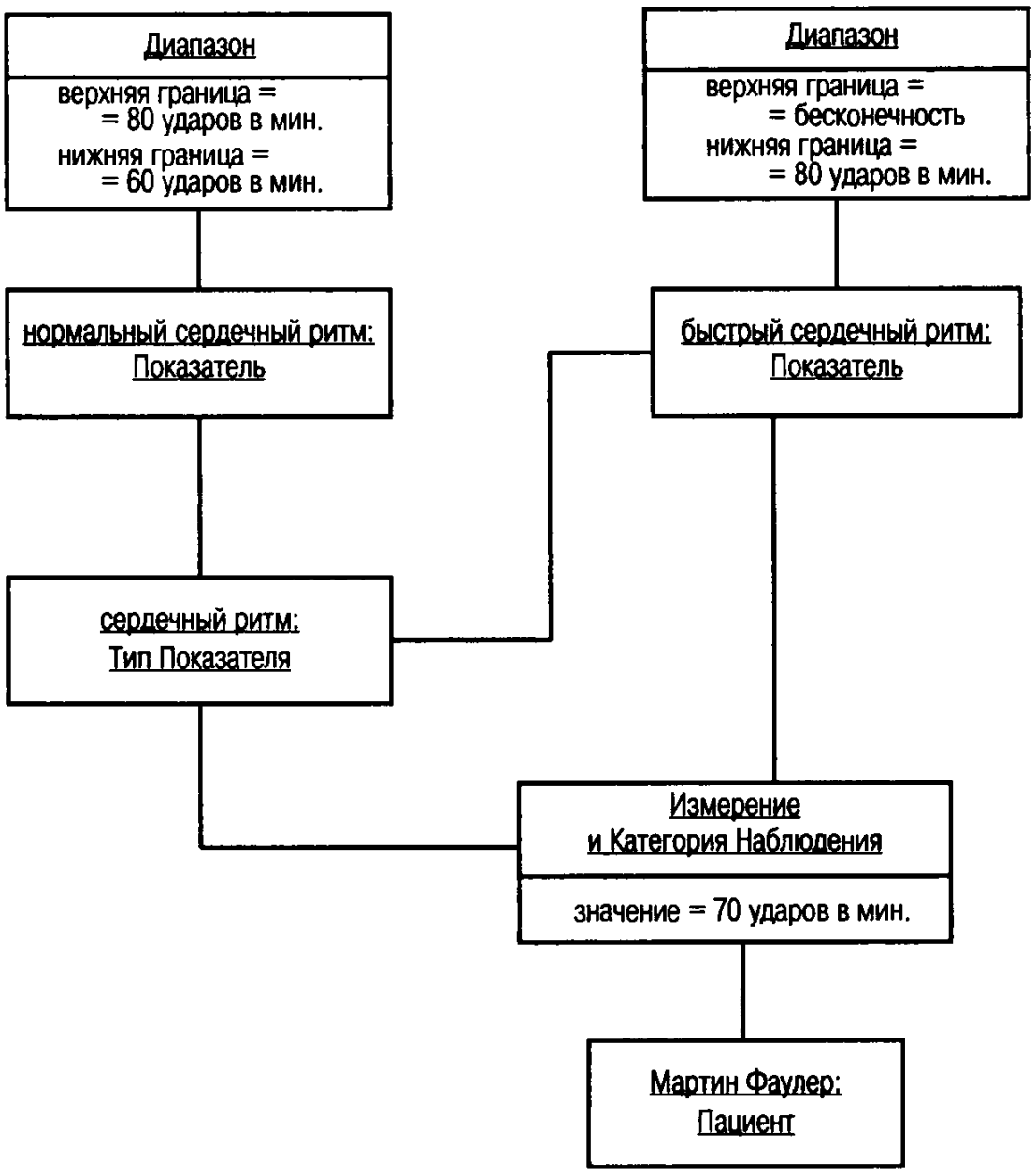


Рис. 11.3. *Другая диаграмма объектов наблюдения пациента*

Проблема, связанная с Наблюдением, заключается в том, что Наблюдение одновременно может быть Категорией Наблюдения и Измерением (рис. 11.3). В языке Java, как и в большинстве других языков программирования, можно определить только одну классификацию. Я решил эту проблему, допустив, что любое Наблюдение должно иметь ассоциированный с ним Показатель, который позволяет классу Наблюдение эффективно реализовывать как понятие Наблюдение, так и понятие Категория Наблюдения.

Хотя эти решения далеки от совершенства, тем не менее, они позволяют выполнить намеченную работу. Не пытайтесь создать программное обеспечение, которое в точности отражало бы концептуальную точку зрения. Напротив, следует придерживаться не буквы, а духа концептуальной модели, учитывая при этом ограничения имеющихся средств реализации.

Наблюдение пациента: модель спецификации

На рис. 11.4 отражены модификации, которые внесены в модель предметной области, чтобы учесть некоторые из факторов, связанных с языком реализации.

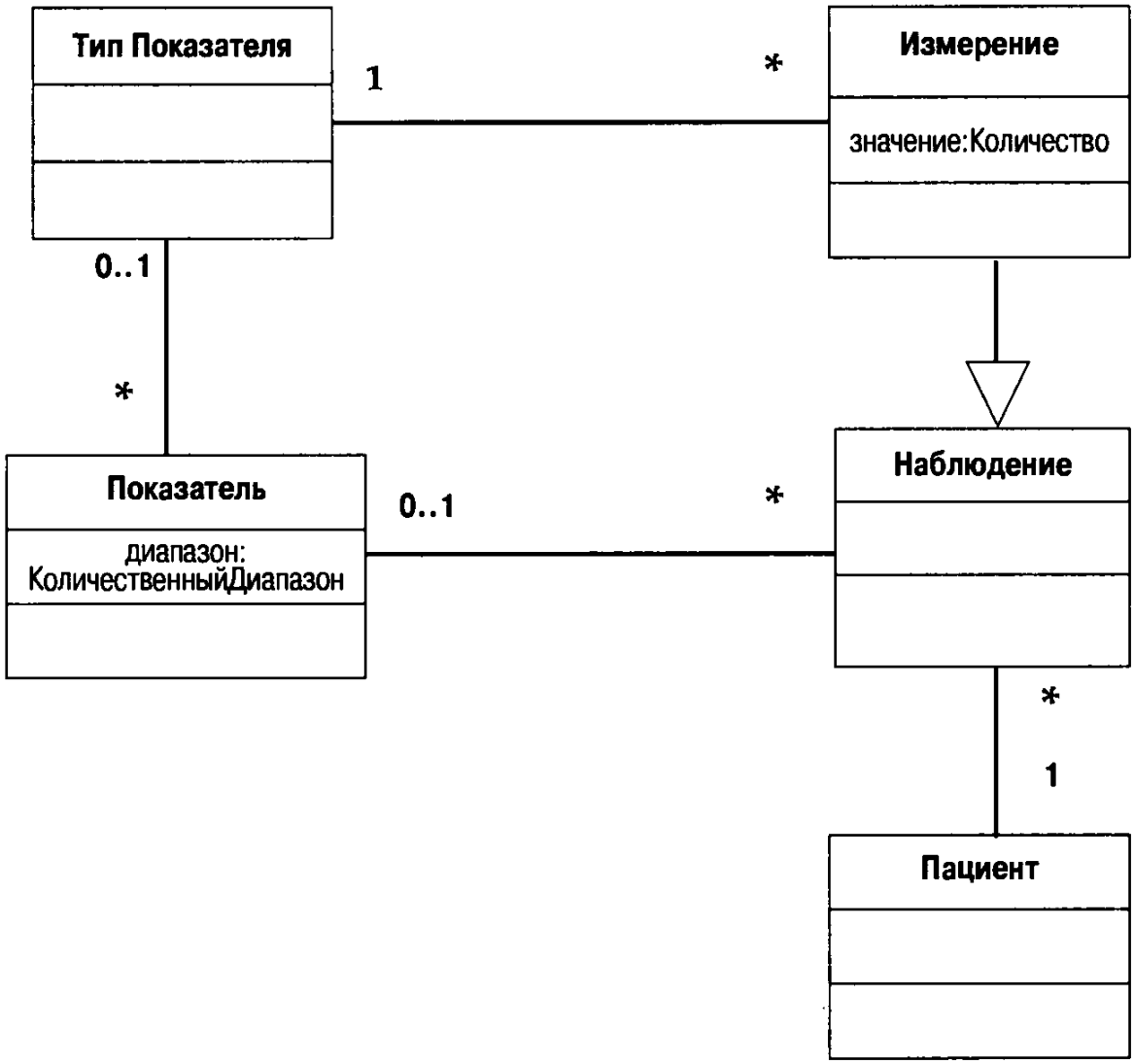


Рис. 11.4. Модель наблюдения пациента уровня спецификации

Модель наблюдения пациента представлена здесь с точки зрения спецификации. На ней указаны скорее интерфейсы классов, чем сами классы. Можно было бы поддерживать эту концептуальную модель и далее, но более вероятно, что начиная с этого момента, я буду работать только с моделью спецификации. Я не стараюсь поддерживать слишком много моделей. Мое личное правило состоит в следующем: если я не могу поддерживать модель посредством внесения в нее необходимых обновлений, то она отправляется в корзину. (Я знаю, что я к тому же ленив!)

Теперь рассмотрим *поведение*, ассоциированное с нашей моделью наблюдения пациента (рис. 11.5).

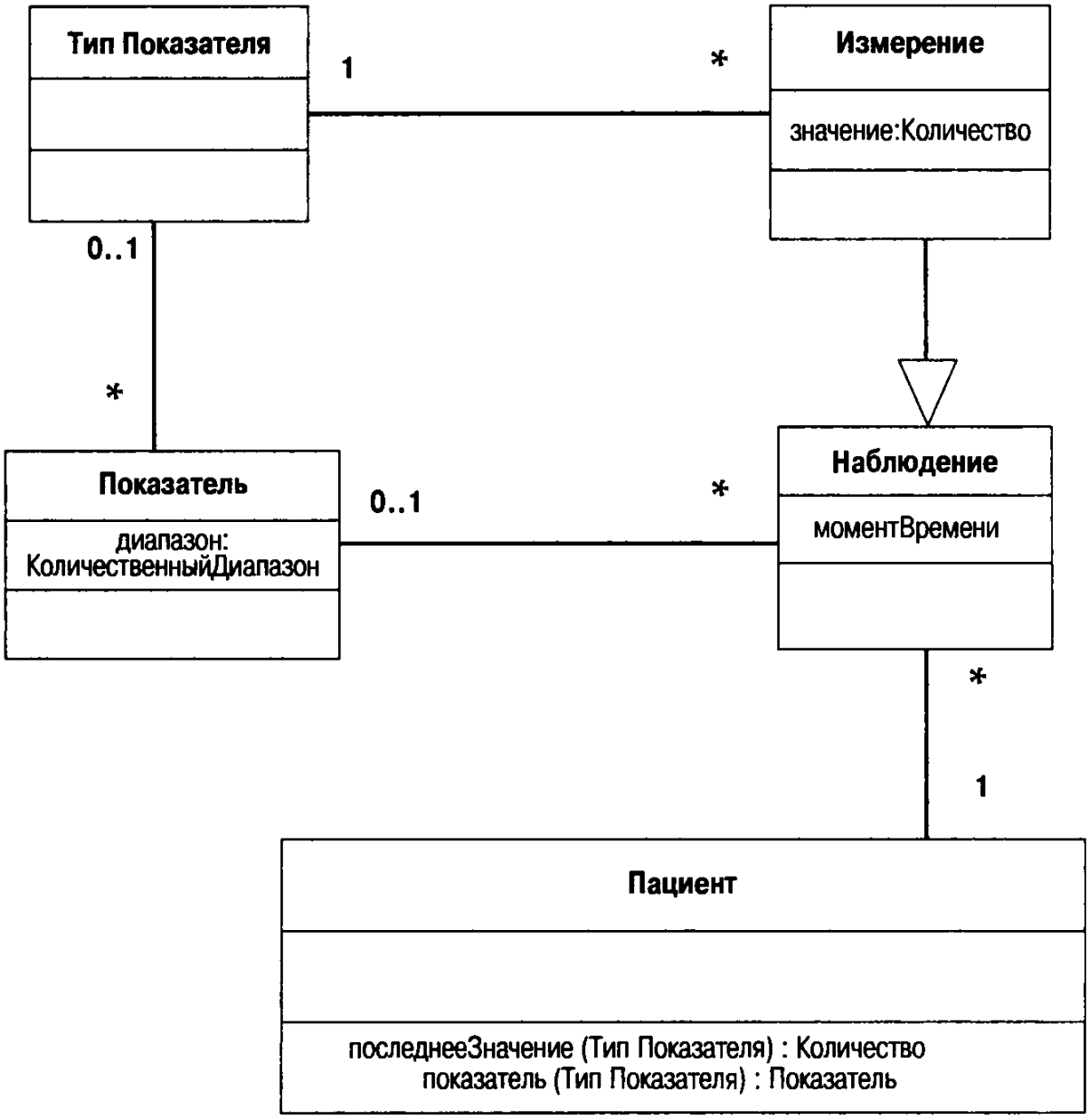


Рис. 11.5. Операции модели наблюдения пациента

Первый сценарий запрашивает последние данные о сердечном ритме пациента. В связи с этим возникает первый вопрос: кто должен нести ответственность за выполнение этого запроса? Естественным ответом представляется Пациент. Пациент должен просмотреть все свои наблюдения, выбрать из них измерения с Типом Показателя «сердечный ритм» и найти среди них самое последнее значение. Чтобы сделать это, необходимо добавить в Измерение момент времени. Поскольку аналогичные рассуждения могут быть применены и к другим наблюдениям, я также добавлю момент времени и в Наблюдение.

Пациент имеет такую же ответственность по отношению к Показателю: Найти последнюю Категорию Наблюдения, которая обладает Показателем для данного Типа Показателя.

На рис. 11.5 показаны те операции, которые были мною добавлены к классу Пациент, чтобы отразить в модели все сказанное выше.

Не пытайтесь слишком тщательно определять операции, если на данный момент они не являются совершенно очевидными. Самое важное сейчас – это установить ответственность. Если вы можете облечь ее в форму операции, то это прекрасно; в противном случае для описания ответственности вполне достаточно и короткой фразы.

Обновление измерения уровня сознания пациента включает создание нового Наблюдения соответствующего Показателя. Выполняя эти действия, пользователь обычно предпочитает выбрать Показатель из некоторого «всплывающего» списка. Это можно реализовать, связав объекты Показатель с конкретным Типом Показателя, поскольку данная ответственность подразумевается ассоциацией между ними.

Добавляя некоторое измерение, тем самым мы должны создать новое Измерение. Некоторые дополнительные сложности вытекают из того факта, что это новое Измерение должно установить, существует ли Показатель, который может быть для него назначен. В этом случае Измерение может запросить ассоциированный с ним Тип Показателя, имеется ли соответствующий Показатель.

Таким образом, между данными объектами существует некоторая кооперация. Она может быть хорошо представлена с помощью диаграммы последовательности (рис. 11.6).

Должны ли вы строить все эти диаграммы?

Вовсе нет. Многое зависит от ваших способностей к визуализации моделей и от того, насколько легко вам работать с выбранным языком программирования. Что касается языка Smalltalk, то зачастую писать код так же легко, как и разрабатывать диаграммы. Если же программа разрабатывается на языке C++, то диаграммы приносят больше пользы.

Диаграммы не должны быть произведениями искусства. Я обычно изображаю их в виде схем на листе бумаги или небольшой доске. Я прибегаю к помощи графического редактора (или CASE-средства) только тогда, когда считаю целесообразным тратить усилия на поддержание их в состоянии внесения возможных обновлений, поскольку они помогают внести ясность в поведение классов. На этой стадии проекта я могу также пользоваться CRC-карточками (см. главу 5) в дополнение или вместо тех диаграмм, которые описаны в данной главе.

Переход к кодированию

Теперь мы можем обратиться к рассмотрению фрагментов программного кода, реализующего те идеи, которые обсуждались в предыдущих разделах. Я начну с Типа Показателя (Phenomenon Type) и Показателя (Phenomenon), поскольку они довольно тесно взаимосвязаны.

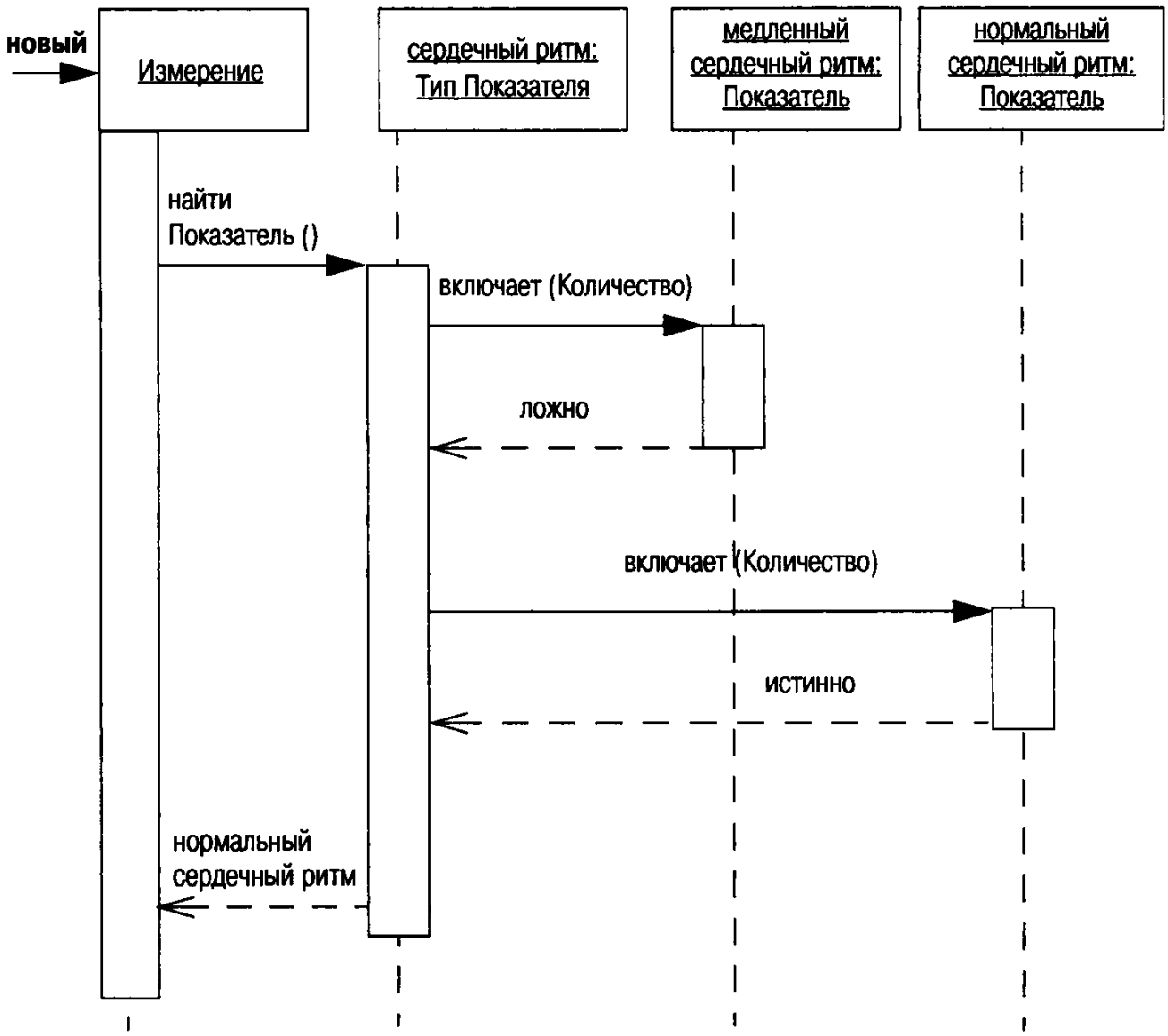


Рис. 11.6. Диаграмма последовательности наблюдения пациента

Первое, на что следует обратить внимание, – это ассоциация между ними: должен ли интерфейс допускать навигацию в обоих направлениях? По моему мнению, в данном случае именно так и должно быть, поскольку при этом оба направления будут одинаково важными и, в любом случае, эти понятия тесно связаны друг с другом. На практике данную ассоциацию удается реализовать с помощью указателей в обоих направлениях. Однако эту ассоциацию я сделаю неизменяемой, поскольку даже при обновлении данных соответствующие объекты не будут модифицироваться часто, а если и будут, мы всегда можем создать их заново.

Некоторые разработчики испытывают трудности при работе с двунаправленными связями. Я не думаю, что могут возникнуть какие-либо проблемы, если обеспечить такую реализацию, при которой один класс несет полную ответственность за поддержание данной связи в актуальном состоянии, пользуясь при необходимости специальным «дружественным» или вспомогательным методом.

Давайте рассмотрим некоторые определения.

```
public class PhenomenonType extends DomainObject {
 public PhenomenonType (String name){
  super (name);
 };

 void friendPhenomenonAdd (Phenomenon newPhenomenon) {
  \\ ОГРАНИЧЕН: используется только Показателем
  _phenomena.addElement (newPhenomenon);
 };

 public void setPhenomena (String[] names) {
  for (int i = 0; i < names.length; i++)
   new Phenomenon (names[i], this);
 };

 public Enumeration phenomena() {
  return _phenomena.elements();
};

 private Vector _phenomena = new Vector();

 private QuantityRange _validRange;
}
```

Здесь используется соглашение, в соответствии с которым ко всем полям перед их именем добавляется символ подчеркивания. Это помогает избежать недоразумений, связанных с именами.

```
public class Phenomenon extends DomainObject {
 public Phenomenon (String name, PhenomenonType type) {
  super (name);
  _type = type;
  _type.friendPhenomenonAdd (this);
 };
 public PhenomenonType phenomenonType() {
  return _type;
 };

 private PhenomenonType _type;
 private QuantityRange _range;
}

package observations;

public class DomainObject {
  public DomainObject (String name) {
  _name = name;
};

  public DomainObject()  {};
```

```
public String name() {
   return _name;
};

public String toString() {
  return _name;
};

protected String _name = "no name";
}
```

Я добавил класс с именем DomainObject (Объект Предметной Области), который располагает информацией об именах и сможет реализовать любое другое поведение, которое потребуется от всех моих классов предметной области.

Теперь я могу описать эти объекты с помощью следующего кода:

```
PhenomenonType sex =
  new PhenomenonType("gender").persist();
String[] sexes = {"male", "female"};
sex.setPhenomena(sexes);
```

Операция **persist()** сохраняет Тип Показателя в специальном объекте-реестре, чтобы впоследствии его снова можно было получить с помощью статического метода **get()**. Детали реализации этого я опущу.

Теперь рассмотрим фрагмент программного кода, который реализует ввод новых наблюдений пациента. В данном случае мне не нужно, чтобы все ассоциации были двунаправленными. Я связал пациента с некоторой совокупностью наблюдений, поскольку сами наблюдения используются только в контексте отдельных пациентов.

```
public class Observation extends DomainObjest {
 public Observation (Phenomenon relevantPhenomenon,
  Patient patient, Date whenObserved) {
  _phenomenon = relevant Phenomenon;
   patient.observationsAdd(this);
  _whenObserved = whenObserved;
};

private Phenomenon _phenomenon;

private Date _whenObserved;
}

public class Patient extends DomainObject {
  public Patient(String name) {
  super(name);
};

void observationsAdd(Observation newObs) {
```

```
    _observations.addElement(newObs);
  };

  private Vector _observations = new Vector();
  }
```

С помощью следующего фрагмента кода я могу создавать наблюдения.

```
new Patient("Adams").persist();
new Observation(PhenomenonType.get("gender"),
  phenomenonNamed("male"), Patient.get("Adams"),
  new Date (96, 3, 1) );
class PhenomenonType {
  public Phenomenon phenomenonNamed(String name) {
  Enumeration e = phenomena();
  while (e.hasMoreElements() )
  {
   Phenomenon each = (Phenomenon)e.nextElement();
   if (each.name().equals(name))
    return each;
  };

  return null;
}
```

После создания наблюдений необходимо иметь возможность поиска самого последнего показателя.

```
class Patient
 public Phenomenon phenomenonOf
  (PhenomenonType phenomenonType)
 {
  return (latestObservation(phenomenonType) ==
   null ? new Nu11Phenomenon() :
   latestObservation(phenomenonType).phenomenon() );
 }

 private Obsertation
  latestObservation(PhenomenonType value) {
  return latestObservationln(observationsOf(value) );
}

private Enumeration
 observationsOf(PhenomenonType value) {
  Vector result = new Vector();
  Enumeration e = observations();
  while (e.hasMoreElements() )
  {
   Observation each = (Observation) e.nextElement();
   if (each.phenomenonType() = = value)
    result.addElement(each);
 };
```

```
 return result.elements();
}
private Observation latestObservationIn
 (Enumeration observationEnum)
{
 if (!observationEnum.hasMoreElements() )
  return null;
 Observation result =
  Observation)observationEnum.nextElement();
 if (!observationEnum.hasMoreElements() )
  return result;
 do
 {
  Observation each =
   (Observation)observationEnum.nextElement();
  if (each.whenObserved().
   after(result.whenObserved() ) )
    result = each;
 }

 while (observationEnum.hasMoreElements() );

  return result;
}

class Observation
 public PhenomenonType phenomenonType() {
  return _phenomenon.phenomenonType() ;
}
```

Для реализации этого можно объединить нескольких методов в один. Для иллюстрации можно было бы построить диаграмму, но я не буду этого делать. Способ, которым я декомпозирую метод, больше относится к реорганизации (см. главу 2), чем к уже выполненному проектированию.

Теперь можно рассмотреть добавление поведения для измерений.

Во-первых, давайте посмотрим на определение класса Измерение (Measurement) и его конструктор.

```
public class Measurement extends Observation {
 public Measurement(Quantity amount, PhenomenonType phenomenonType,
  Patient patient, Date whenObserved) {
   initialize (patient, whenObserved);
   _amount = amount ;
   _phenomenonType = phenomenonType;
 };

 public PhenomenonType phenomenonType() {
  return _phenomenonType;
 };
```

```
public String toString() {
 return _phenomenonType + ": " + _amounts;
};

private Quantity _amount;

private PhenomenonType _phenomenonType;
}

class Observation
 protected void initialize(Patient patient, Date whenObserved) {
  patient.observationsAdd(this);
  _whenObserved = whenObserved;
}
```

Следует заметить, что диаграмма классов служит хорошей отправной точкой для разработки данного фрагмента кода.

И снова нам требуется самое последнее измерение.

```
Class Patient
 public Quantity latestAmountOf(phenomenonType value) {
  return ((latestMeasurement(value) == null) ) ?
  new NullQuantity():latestMeasurement(value).amount();
 }

 private Measurement
  latestMeasurement(PhenomenonType value) {
   if (latestObservation(value) == null)
    return null;
   if (!latestObservation(value).isMeasurement() )
    return null;
   return (Measurement)latestObservation(value);
 }
```

В каждом их этих случаев диаграмма классов определяет основную структуру, мы же только добавляем поведение, чтобы реализовать интересующие нас запросы.

На данной стадии можно описать наше положение с помощью диаграммы классов уровня спецификации, изображенной на рис. 11.7.

Посмотрим, каким образом данная диаграмма акцентирует внимание на интерфейсе, а не на реализации. Роль от Типа Показателя к Показателю моделируется как некая квалифицированная роль, поскольку это главный интерфейс класса Тип Показателя. Аналогично показана связь Наблюдения с Типом Показателя, поскольку и здесь существует интерфейс, хотя имеется только одно измерение с прямым указателем на Показатель.

Глядя на эту диаграмму, мы можем сделать вывод, что единственное различие между классами Измерение и Наблюдение заключается в том,

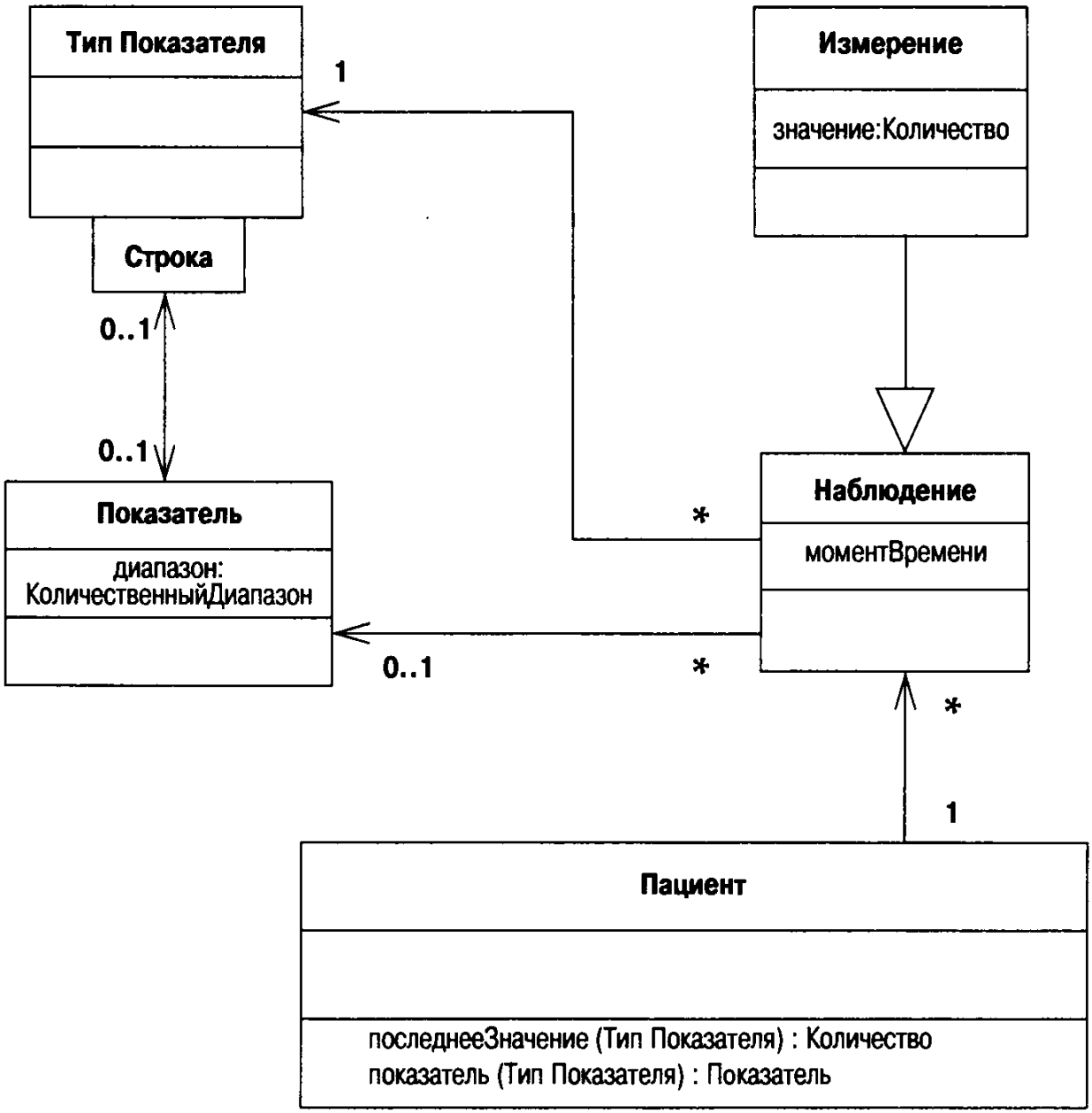


Рис. 11.7. *Другая модель спецификации наблюдения пациента*

что Измерение обладает количеством. Класс Измерение можно совсем удалить из модели спецификации, предположив, что любое наблюдение обладает количеством (которое может иметь неопределенное значение null).

Мы могли бы все же оставить отдельный класс Измерение с полями *величина* и *тип показателя*, но при этом никто за пределами данного пакета не будет знать о существовании данного класса. Чтобы обеспечить создание соответствующего класса, нам может понадобиться добавить методы образца «Фабрика» (Factory) (Гамма и др., 1995 [20]) либо в класс Наблюдение, либо в класс Пациент.

Я оставлю это дополнение в качестве упражнения для читателей и перейду к автоматическому связыванию Показателя с Измерением.

Этот общий процесс изображен на рис. 11.7.

Сначала необходимо добавить метод вызова в конструктор класса Измерение.

```
Class Measurement
 public Measurement (Quantity amount, PhenomenonType phenomenonType,
 Patient patient, Date whenObserved)
  initialize (patient, whenObserved);
  _amount = amount ;
  _phenomenonType = phenomenonType;
  _phenomenon = calculatePhenomenonFor(_amount);
```

Далее эта задача делегируется классу Тип Показателя.

```
Class Measurement
 public Phenomenon calculatePhenomenonFor(Quantity arg)
 {
  return _phenomenonType.phenomenonIncluding(arg);
 }
```

Далее по очереди запрашивается каждый показатель.

```
Class PhenomenonType
public Phenomenon phenomenonIncluding (Quantity arg) {
 Enumeration e = phenomena();
 while (e.hasMoreElements() )
 {
  Phenomenon each = (Phenomenon) e.nextElement();
  if (each.includes(arg))
   return each;
 };

 return null;
}
Class Phenomenon
 public boolean includes (Quantity arg) {
  return (_range == null ? false:_range.includes (arg));
}
```

Данный код достаточно явно следует из диаграммы последовательности. На практике я обычно пользуюсь диаграммой последовательности, чтобы только приближенно представить взаимодействие, а затем по мере кодирования вношу в него необходимые изменения. Если взаимодействие имеет существенное значение, я поддерживаю диаграмму последовательности в активном состоянии как часть моей документации. Если же я сочту, что диаграмма последовательности не вносит никакой дополнительной ясности в программный код, то сдам ее в архив.

Этот небольшой пример показывает, как использовать языка UML в совокупности с языком программирования, однако при этом он дает и общее представление о процессе. Нет необходимости черезчур формально подходить к выполнению проекта и пытаться использовать аб-

солютно все возможности языка UML. Вполне достаточно пользоваться только теми из них, которые представляются вам полезными.

Схематическое представление проекта с помощью диаграммы классов и диаграммы взаимодействия помогает привести мысли в порядок и существенно облегчает процесс кодирования. Я рассматриваю эти схемы в качестве первых прототипов. В дальнейшем эти диаграммы вовсе не обязательно поддерживать, однако может оказаться, что они будут облегчать понимание программного кода вам самим и другим разработчикам.

Нет необходимости использовать какое-то изысканное и дорогостоящее CASE-средство. Обычной доски и несложного графического редактора на компьютере может оказаться вполне достаточно. Конечно, существуют полезные CASE-средства, и если вы участвуете в крупномасштабном проекте, то следует рассмотреть возможность применения какого-либо из них.

Если вы решите использовать какое-либо CASE-средство, сравните его возможности с простым графическим редактором и текстовым процессором. (Просто поразительно, как много можно сделать с помощью таких средств, как Visio и Word.) Если средство обладает возможностью генерации кода, то следует с большим вниманием присмотреться к тому, каким образом оно это делает. Возможность генерации кода CASE-средством привносит крайне специфическую интерпретацию диаграмм, которая может повлиять на их смысл и ваш способ построения.

Дополнительную информацию по данному примеру можно найти на моем сайте. В приведенной там версии примера более детально рассмотрены вопросы взаимосвязи данной модели с пользовательским интерфейсом.

A

Средства и их использование

Средство	Назначение
Диаграмма деятельности	Показывает поведение вместе со структурой управления. Может изображать множество объектов в нескольких вариантах использования, множество объектов в единственном варианте использования или реализацию метода. Предоставляет возможность определять параллельные процессы.
Диаграмма Классов	Показывает статическую структуру понятий, типов и классов. Понятия отражают представление пользователей о реальном мире; типы отражают интерфейсы компонентов программного обеспечения; классы отражают реализацию компонентов программного обеспечения.
CRC-карточки	Помогают выделить сущность назначения класса. Полезное средство при поиске ответа на вопрос: как реа-лизовывать вариант использования. Следует применять при возникновении проблем с деталями реализации, а также при изучении объектного подхода к проектированию.
Диаграмма развертывания	Показывает физическое размещение компонентов на узлах аппаратуры.

Средство	Назначение
Проектироваиие по контракту	Обеспечивает строгое определение назначения операции и допустимого состояния класса. Кодирует данное определение в классе с целью расширения возможностей отладки.
Диаграмма взаимодействия	Показывает кооперацию нескольких объектов в рамках одного варианта использования.
Диаграмма пакетов	Показывает группы классов и зависимости между ними.
Образцы	Являются полезным средством для анализа, проектирования и кодирования. Представляют собой хорошие примеры для обучения. Могут служить начальной точкой для проектирования.
Реорганизация	Помогает вносить изменения в работающую программу с целью улучшения ее структуры. Следует применять при необходимости тщательного проектирования программного кода.
Диаграмма состояний	Показывает особенности поведения единственного объекта в нескольких вариантах использования.
Вариант использования	Облекает пользовательские требования в законченную форму. Планирование фазы построения осуществляется с учетом реализации на каждой итерации нескольких вариантов использования. Является основой для тестирования системы.

B

Отличия версий языка UML

Когда вышло в свет первое издание этой книги, язык UML имел еще версию 1.0. С ее появлением многие элементы языка UML обрели устоявшуюся терминологию, а консорциум OMG приобрел официальное признание. С тех пор версия языка UML пересматривалась несколько раз. В этом приложении описываются все существенные изменения языка UML с момента появления версии 1.0, что позволит учесть их, если вы пользуетесь первым изданием. Эволюция языка UML потребовала обновления книги, и второе издание содержит материал, отражающий ситуацию на момент его написания.

Эволюция языка UML

Первой общедоступной версией языка UML был Унифицированный метод версии 0.8, который был представлен на конференции OOPSLA, состоявшейся в октябре 1995 года. Унифицированный метод был разработан Г. Бучем и Д. Рамбо (к этому моменту А. Джекобсон еще не был сотрудником компании Rational). В 1996 году компания Rational выпустила версии 0.9 и 0.91, в работе над которыми принимал участие Джекобсон. После выхода этой последней версии название метода изменилось на UML.

В январе 1997 года компания Rational представила на рассмотрение инициативной группы анализа и проектирования из OMG версию 1.0 языка UML. В последующем компания Rational объединила эту версию UML с другими методами и в сентябре 1997 года предложила в ка-

честве стандарта версию 1.1. В конце 1997 года версия была одобрена консорциумом OMG. Однако при невыясненных обстоятельствах консорциум OMG назвал этот стандарт языка UML версией 1.0. Таким образом, в то время существовали две версии языка UML: версия 1.0 консорциума OMG и версия 1.1 компании Rational, которые не следует путать с версией 1.0 компании Rational. На практике же все разработчики называли этот стандарт версией 1.1.

Версия 1.1 языка UML имеет несколько незначительных отличий от версии 1.0.

В ходе работы над версией 1.1 UML в рамках консорциума OMG была образована инициативная группа по пересмотру языка (Revisoin Task Force, RTF), которую возглавил Крис Кобрин. Задача группы состояла в устранении неточностей языка UML. В июле 1998 внутри консорциума OMG была выпущена версия 1.2. Появление последней считается внутренним делом OMG, поскольку официальным стандартом UML оставалась версия 1.1. Поэтому версию 1.2 можно считать бета-версией языка UML. Однако в действительности эти изменения едва заметны, поскольку были опубликованы только изменения в стандарте языка UML: фиксированные типы, грамматические ошибки и т. п.

Более значительные изменения коснулись версии 1.3, которые заметно уточнили терминологию в отношении вариантов использования и диаграмм деятельности. Позже в 1999 году были опубликованы две книги «трех друзей»: «Руководство пользователя» [6] и «Справочник пользователя» [37], которые отразили изменения в версии 1.3 до публикации официальных документов по этой версии языка UML, что привело к некоторым недоразумениям.

В апреле 1999 года инициативная группа RTF представила консорциуму OMG версию 1.3 в качестве нового официального стандарта языка UML. После чего эта группа взяла на себя дальнейшее развитие языка UML и рассмотрение его последующих обновлений. Эта информация известна мне на момент написания книги; последние сведения по этим вопросам можно найти на моей домашней страничке в Интернете.

Изменения в первом издании книги

В процессе эволюции языка UML я пытался учесть его изменения, что привело к появлению нескольких вариантов книги «UML в кратком изложении». При этом я пользовался хорошей возможностью исправить ошибки и сделать изложение более ясным.

Хотя в основу настоящего второго издания положена версия 1.3, я не могу утверждать, что последующие варианты этой книги не будут содержать каких-либо неожиданных обновлений. Так, в первом варианте приложения упоминалась версия 1.4, но в конце концов инициатив-

ная группа RTF решила не изменять номер версии языка UML, поэтому она так и осталась версией 1.3.

Далее в этом приложении внимание сосредоточено на двух основных отличиях в языке UML, имевших место при изменениях версий с 1.0 на 1.1 и с 1.2 на 1.3. Мне бы не хотелось обсуждать в деталях все произошедшие изменения, поэтому остановлюсь лишь на тех из них, которые каким-то образом затрагивают материал книги «UML в кратком изложении» или относятся к важным свойствам языка UML, рассмотренным в первом издании.

Продолжая следовать стилю изложения первого издания, я рассмотрю основные элементы языка UML и особенности применения языка UML при выполнении реальных проектов. Как и ранее, выбор материала и соответствующие рекомендации основаны на моем собственном опыте. Если обнаружится противоречие между моим изложением и официальной документацией по языку UML, следует придерживаться официальной документации. (Однако мне бы хотелось об этом знать, чтобы впоследствии внести соответствующие исправления.)

Я также воспользуюсь представленной возможностью отметить те или иные важные ошибки или погрешности предыдущих вариантов книги. Спасибо читателям, которые сообщили о них.

Отличия версий 1.0 и 1.1 языка UML

Тип и класс реализации

В первом издании «UML в кратком изложении» я рассмотрел различные точки зрения на разработку и те возможные изменения, которые произойдут в результате совершенствования способов изображения и интерпретации моделей, в частности, диаграмм классов.

Эти обстоятельства нашли отражение в языке UML, поскольку теперь утверждается, что все классы на диаграмме классов могут быть определены либо как типы, либо как классы реализации.

Класс реализации соответствует некоторому классу в контексте программы, которая вами разрабатывается. **Тип** является более расплывчатым понятием; он представляет некоторую абстракцию, которая в меньшей степени касается реализации. Это может быть тип CORBA, описание класса с точки зрения спецификации или некоторая концептуальная модель. При необходимости можно определить дополнительные стереотипы, чтобы в последующем различать эти понятия.

Можно установить, что для отдельной диаграммы все классы обладают некоторым особым стереотипом. Это может произойти в том случае, когда диаграмма отражает отдельную точку зрения. При этом точка зрения реализации предполагает использование классов реали-

зации, а концептуальная точка зрения и точка зрения спецификации предполагают использование типов.

Если некоторый класс реализации реализует один или несколько типов, то это может быть показано с помощью отношения реализации.

Между типом и интерфейсом существует различие. Предполагается, что некоторый интерфейс должен непосредственно соответствовать интерфейсу в стиле CORBA или COM. В этом случае интерфейсы имеют только операции и не имеют атрибутов.

Для класса реализации может быть использована только единственная статическая классификация, однако для типов можно использовать множественную и динамическую классификацию. (Я думаю, что причиной этого является то обстоятельство, что основные объектно-ориентированные языки поддерживают единственную статическую классификацию. Если в один прекрасный день вы будете пользоваться языком, который поддерживает множественную или динамическую классификацию, то это ограничение может быть снято.)

Ограничения полного и неполного обобщения

В предыдущих выпусках *UML в кратком изложении* было отмечено, что ограничение {полный} ({complete}) для некоторого обобщения устанавливает, что все экземпляры супертипа должны быть также экземпляром некоторого подтипа в данном разбиении. Вместо этого в языке UML версии 1.1 определено ограничение {полный}, которое указывает лишь на то, что соответствующее разбиение отражает все подтипы. А это совсем не то же самое. Мною было обнаружено множество несоответствий в интерпретации этого ограничения, поэтому вам следует обратить на это внимание.

Если вы хотите, чтобы все экземпляры супертипа были экземпляром одного из подтипов, то во избежание недоразумений логично использовать другое ограничение. Лично я использую в этом случае {обязательно} ({mandatory}).

Композиция

Использование композиции в языке UML версии 1.0 означает, что эта связь неизменна (или постоянна) по крайней мере для однозначных компонентов. Это ограничение больше не является частью определения композиции.

Неизменность и постоянство

Язык UML определяет ограничение {постоянный} ({frozen}) для указания неизменяемости ролей ассоциации. Как определено в настоящее время, это ограничение не может быть применено к атрибутам или

классам. В своей текущей работе вместо неизменяемости я использую термин постоянный, тем самым я могу применять это ограничение к ролям ассоциаций, классам и атрибутам.

Обратные сообщения на диаграмме последовательности

В языке UML версии 1.0 обратное сообщение или возврат на диаграмме последовательности вместо сплошной треугольной стрелки стало обозначаться обычной стрелкой (см. предыдущее издание). Это привело к некоторым проблемам, поскольку данное различие трудно уловимо и легко приводит к недоразумениям.

Язык UML версии 1.1 для изображения возвратов использует пунктирную линию со стрелкой, что мне больше нравится, поскольку делает возвраты намного более очевидными. (Именно поэтому в своей книге «Анализ образцов» [18] я использовал пунктирные возвраты, что представляется мне весьма важным.) Для последующего применения возвратов можно назначить им имена вида *enoughStock:=check( )*.

Использование термина «Роль»

В языке UML версии 1.0 термин **роль** в основном указывал направление некоторой ассоциации (см. предыдущее издание). Язык UML версии 1.1 рассматривает данное определение как **роль ассоциации**. Помимо нее существует **роль кооперации**, то есть роль, которую исполняет некоторый экземпляр класса в кооперации.

Многие разработчики по-прежнему используют термин роль в смысле направления ассоциации, хотя конец ассоциации является официальным термином языка UML.

Отличия версий 1.2 (и 1.1) и 1.3 языка UML

Варианты использования

Изменения относительно вариантов использования заключаются в добавлении новых отношений между вариантами использования.

В языке UML версии 1.1 имелись только два отношения между вариантами использования: «использует» и «расширяет», каждое их которых является стереотипом обобщения. В версии 1.3 определены три отношения:

- Конструкция «включает» является стереотипом зависимости. Она означает, что выполнение одного варианта использования включает в себя другой вариант использования. Обычно это отношение встречается в ситуации, когда несколько вариантов использования имеют общие этапы или части. Включаемый вариант использова-

ния может предоставлять другим некоторое общее поведение. В качестве примера можно рассмотреть банкомат ATM, в контексте которого оба варианта использования «Выдать деньги по карточке» и «Осуществить оплату по карточке» используют вариант «Проверить подлинность клиента». Это отношение в общем случае заменяет применение стереотипа «использует».

- Обобщение варианта использования означает, что один вариант использования является вариацией другого. Таким образом, можно иметь один вариант использования для «Выдать деньги по карточке» (базовый вариант использования) и другой вариант использования для ситуации, когда выдача денег невозможна по причине отсутствия средств на счету клиента. Отказ от выплаты денег можно представить в виде отдельного варианта использования, который уточняет базовый вариант использования. (Кроме того, можно определить еще и дополнительный сценарий для варианта использования «Выдать деньги по карточке».) В этом случае специальный вариант использования, подобно рассмотренному выше, может изменить какой-либо аспект базового варианта использования.
- Конструкция «расширяет» является стереотипом зависимости. Она обеспечивает более управляемую форму расширения по сравнению с отношением обобщения. В этом случае в базовом варианте использования задается несколько точек расширения. Включающий вариант использования может вносить изменения в свое поведение только в этих точках расширения. К примеру, при рассмотрении покупки товара через Интернет можно определить один вариант использования для покупки товара с точками расширения для ввода информации о доставке товара и ввода информации об оплате товара. После чего этот вариант использования может быть расширен для постоянных клиентов, для которых подобная информация может быть получена другим способом.

Существует некоторая путаница насчет старой и новой интерпретаций указанных отношений.

Большинство разработчиков применяют стереотип «использует» в ситуациях, когда версия 1.3 рекомендует указывать стереотип «включает», поскольку для многих из них стереотип «включает» может быть заменен стереотипом «использует». И большинство разработчиков применяют стереотип «расширяет» из версии 1.1 в более широком смысле, предполагая не только отношение «расширяет» из версии 1.3, но также и важнейшую составляющую отношения обобщения в версии 1.3. Поэтому можно считать, что отношение со стереотипом «расширяет» расщепляется в версии 1.3 на два отношения: со стереотипом «расширяет» и обобщение.

Хотя это объяснение охватывает большую часть известных мне приложений языка UML, в настоящее время мне неизвестен строгий и правильный способ использования в них старых отношений. Однако боль-

шинство разработчиков вовсе не пользуются этим строгим определением отношений, поэтому мне не хотелось бы развивать эту тему дальше.

Диаграммы деятельности

С появлением версии 1.2 языка UML осталось всего лишь несколько открытых вопросов относительно семантики диаграмм деятельности. В версии 1.3 на большинство из этих вопросов были даны ответы, которые были закреплены в семантике языка UML.

В отношении условного поведения теперь стало возможным использовать деятельность в форме ромба как для соединения, так и для ветвления. Хотя для описания условного поведения ни ветвления, ни разделения не являются необходимыми, все более общепринятым становится способ изображения, заключающий условное поведение в скобки.

Символ синхронизации в форме черты теперь относится как к **разделению** (когда управление расщепляется), так и к **слиянию** (когда синхронизируемое управление объединяется снова). Однако теперь никаких дополнительных условий на слияние не накладывается. Необходимо лишь придерживаться правил, гарантирующих соответствие разделений и слияний. По существу это означает, что каждое разделение должно иметь соответствующее слияние, которое соединяет все параллельные нити процесса, берущие начало в исходном разделении. Хотя разделения и слияния могут быть вложенными, их можно удалить с диаграммы, если нити соединяют ветвления (или слияния) напрямую.

Слияния могут произойти только тогда, когда все входящие в него нити завершены. Однако можно определить некоторое условие для выходящей из разделения нити. Если это условие не выполняется, то соответствующая нить считается завершенной и может участвовать в слиянии остальных нитей.

Свойство множественной инициализации больше не поддерживается. Вместо него можно определить динамическую параллельность в некоторой деятельности (указывается с помощью символа «*» внутри прямоугольника деятельности). Такая деятельность может выполняться параллельно несколько раз; все ее вызовы должны быть завершены, прежде чем сможет быть выполнен какой-либо выходящий из нее переход. Это в некоторой степени эквивалентно множественной инициализации и подходящему условию синхронизации, хотя и является менее гибким способом.

Хотя эти правила в какой-то степени уменьшают гибкость диаграмм деятельности, однако они гарантируют, что диаграммы деятельности являются поистине частными случаями автоматов. Отношение между диаграммами деятельности и автоматами стало предметом дискуссии инициативной группы RTF. Последующие версии языка UML (после 1.4) вполне могут определить диаграммы деятельности как диаграммы совершенно другой формы.

Библиография

1. Kent Beck: *Smalltalk Best Practice Patterns*. Prentice Hall, 1996.
2. Kent Beck: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.
3. Kent Beck и Ward Cunningham: «A Laboratory for Teaching Object-Oriented Thinking» *Proceedings of OOPSLA 89*. SIGPLAN Notices, Vol. 24, No. 10, pp. 1–6. См. *http://c2.com/doc/oopsla89/paper.html*.
4. Grady Booch: *Object-Oriented Analysis and Design with Applications, Second Edition*. Addison-Wesley, 1994.

   Буч Г. «Объектно-ориентированный анализ и проектирование с примерами приложений на C++». – Пер. с англ. – М.: «Бином»; СПб: «Невский диалект», 1999. – 560 с.
5. Grady Booch: *Object Solutions: Managing the Object-Oriented Project*. Addison-Wesley, 1996.
6. Grady Booch, James Rumbaugh и Ivar Jacobson [three amigos]: *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

   Буч Г., Рамбо Дж., Джекобсон А. «Язык UML. Руководство пользователя». – Пер. с англ. – М.: ДМК, 2000. – 432 с.
7. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad и Michael Stal: P*attern-Oriented Software Architecture: A System of Patterns*. John Wiley & Sons, 1996.
8. Peter Coad and Jill Nicola: *Object-Oriented Programming*. Yourdon, 1993.
9. Peter Coad and Edward Yourdon: *Object-Oriented Analysis*. Yourdon, 1991.
10. Peter Coad and Edward Yourdon: *Object-Oriented Design*. Yourdon, 1991.
11. Peter Coad, David North и Mark Mayfield: *Object Models: Strategies, Patterns and Applications*. Prentice Hall, 1995.

    Коуд П., Норт Д., Мейфилд М. «Объектные модели. Стратегии, шаблоны и приложения». – Пер. с англ. – М.: «Лори», 1999. – 434 с.

12. Alistair Cockburn: *Surviving Object-Oriented Projects*. Addison-Wesley, 1998.

13. Steve Cook и John Daniels: *Designing Object Systems: Object-Oriented Modeling with Syntropy*. Prentice Hall, 1994.

14. James O. Coplien: «A Generative Development Process Pattern Language» In Coplien and Schmidt, 1995, pp. 183–237.

15. James O. Coplien и Douglas C. Schmidt, eds.: *Pattern Languages of Program Design* [PLoPDl]. Addison-Wesley, 1995.

16. Ward Cunningham: «EPISODES: A Pattern Language of Competitive Development.» In Vlissides, Coplien, and Kerth, 1996, pp. 371–388.

17. Bruce Powel Douglass: *Real-Time UML*. Addison-Wesley, 1998.

18. Martin Fowler: *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1997.

19. Martin Fowler: *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999 .

20. Erich Gamma, Richard Helm, Ralph Johnson и John Vlissides [Gang of Four]: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

    Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – СПб: «Питер», 2001. – 368 с.

21. Adele Goldberg и Kenneth S. Rubin: *Succeeding with Objects: Decision Frameworks for Project Management*. Addison-Wesley, 1995.

22. David Harel: «Statecharts: A Visual Formalism for Complex Systems.» *In Science of Computer Programming*, Vol. 8,1987.

23. Ivar Jacobson, Grady Booch и James Rumbaugh [three amigos]: *The Unified Software Development Process*. Addison-Wesley, 1999.

24. Ivar Jacobson, Magnus Christerson, Patrik Jonsson и Gunnar Overgaard: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

25. Ivar Jacobson, Maria Ericsson и Agneta Jacobson: *The Object Advantage: Business Process Reengineering with Object Technology*. Addison-Wesley, 1995.

26. Andrew Koenig и Barbara Moo: *Ruminations on C++; A Decade of Programming Insight and Experience*. Addison-Wesley, 1997.

27. Philippe Kruchten: *The Rational Unified Process: An Introduction*. Addison-Wesley, 1999.

28. Craig Larman: *Applying UML and Patterns*. Prentice Hall, 1998.

    Ларман К. «Применение UML и шаблонов проектирования». – М.: Вильямс, 2001. – 496 с.

29. James Martin и James J. Odell: *Object-Oriented Methods: A Founda-tion (UML Edition)*. Prentice Hall, 1998.

30. Robert Cecil Martin: *Designing Object-Oriented C++ Applications: Using the Booch Method*. Prentice Hall, 1995.

31. Steve McConnell: *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, 1996.

32. Steve McConnell: *Software Project Survival Guide*. Microsoft Press, 1998.

33. Bertrand Meyer: *Object-Oriented Software Construction*. Prentice Hall, 1997.

34. William F. Opdyke: «Refactoring Object-Oriented Frameworks.» Ph.D. thesis. University of Illinois at Urbana-Champaign, 1992. См. *ftp://st.cs.uiuc.edii/pub/papers/refactoring/opdyke-thesis.ps.Z*.

35. Trygve Reenskaug: *Working with Objects*. Prentice Hall, 1996.

36. James Rumbaugh: *OMT Insights*. SIGS Books, 1996.

37. James Rumbaugh, Ivar Jacob son и Grady Booch [three amigosj: *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

38. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy и William Lorenzen: *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

39. Geri Schneider and Jason P. Winters: *Applying Use Cases. A Practical Guide*. Addison-Wesley, 1998.

40. Sally Shiaer и Stephen J. Mellor: *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon, 1989.

41. Sally Shiaer и Stephen J. Mellor: *Object Lifecycles: Modeling the World in States*. Yourdon, 1991.

42. Sally Shiaer и Stephen J. Mellor: «Recursive Design of an Application Independent Architecture.» *JE Software*, Vol. 14, No. 1, 1997.

43. John M. Vlissides, James O. Coplien и Norman L. Kerth, eds.: *Pattern Languages of Program Design 2* [PLoPD21]. Addison-Wesley, 1996.

44. Kim Walden и Jean-Marc Nerson: *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems*. Prentice Hall, 1995.

45. Jos Warmer и Anneke Kleppe: *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

46. Rebecca Wirfs-Brock, Brian Wilkerson и Lauren Wiener: *Designing Object-Oriented Software*. Prentice Hall, 1990.

Алфавитный указатель

Д

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Х

Ш

Э

Я

Диаграмма пакетов

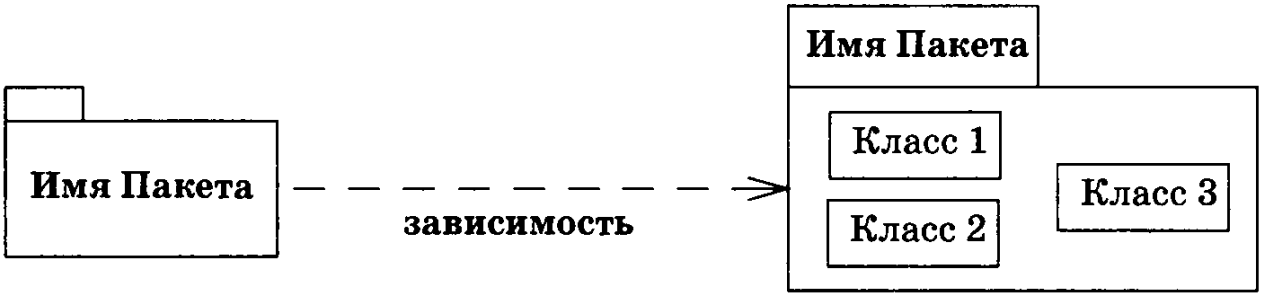


Диаграмма последовательности

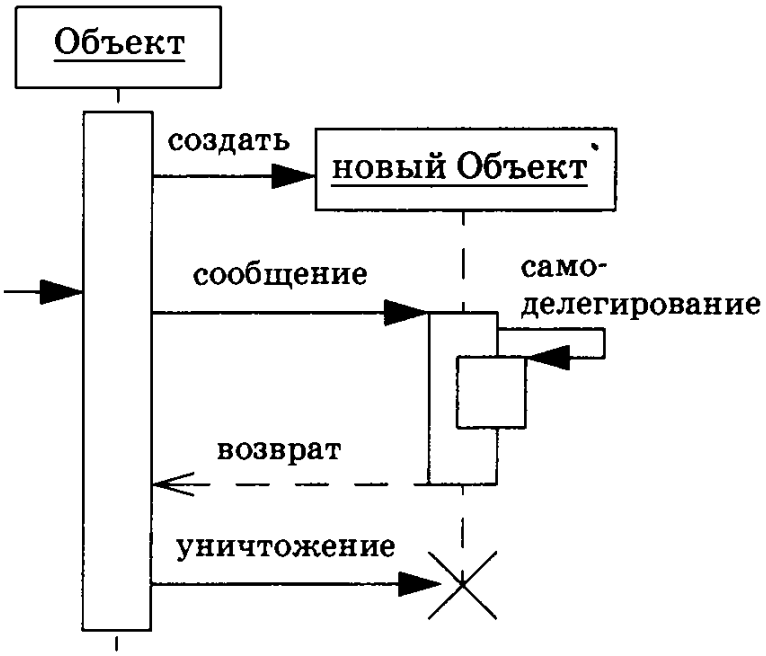


Диаграмма развертывания

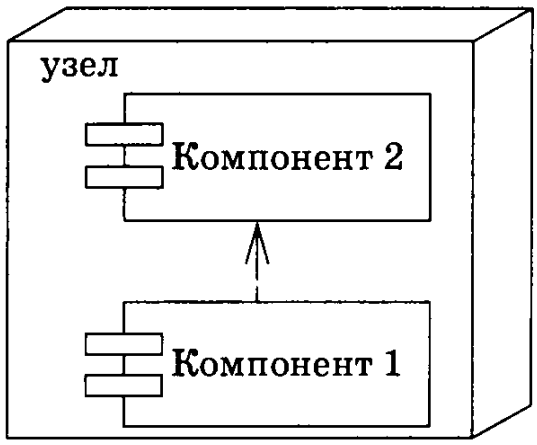


Диаграмма кооперации

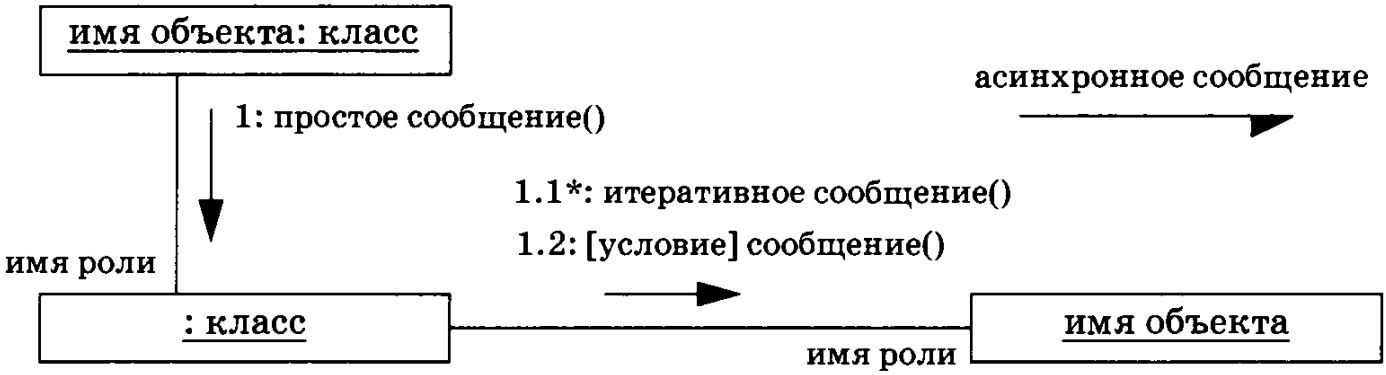


Диаграмма вариантов использования

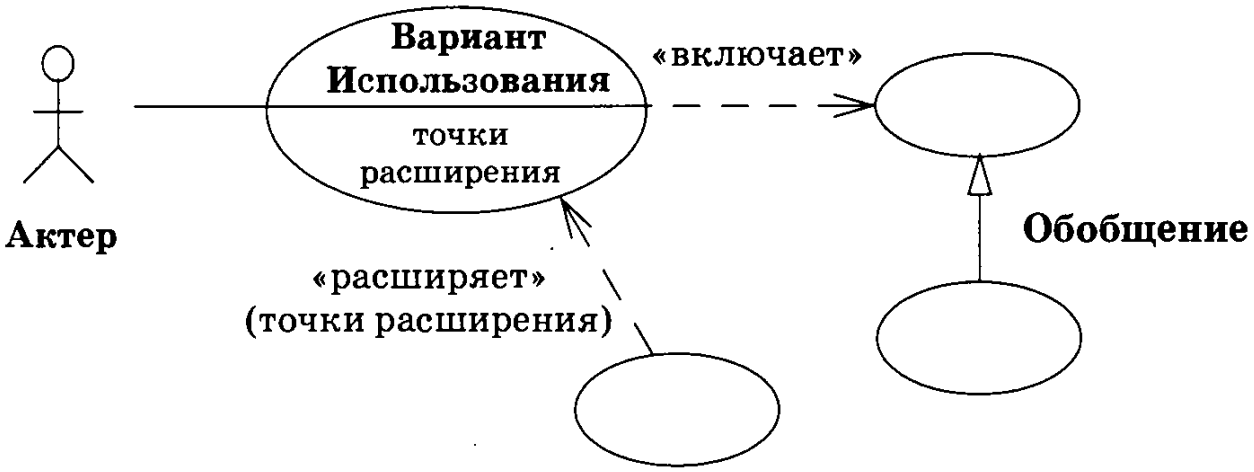
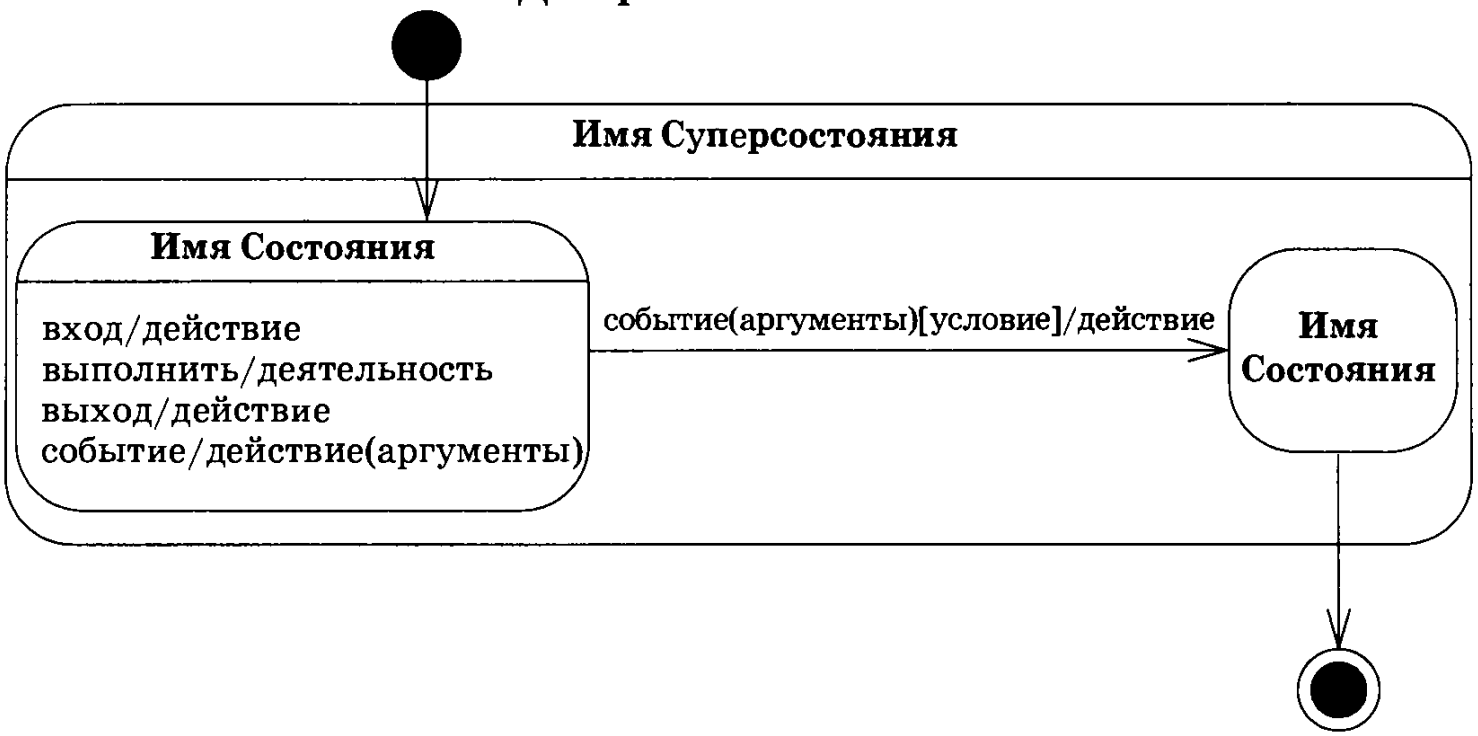
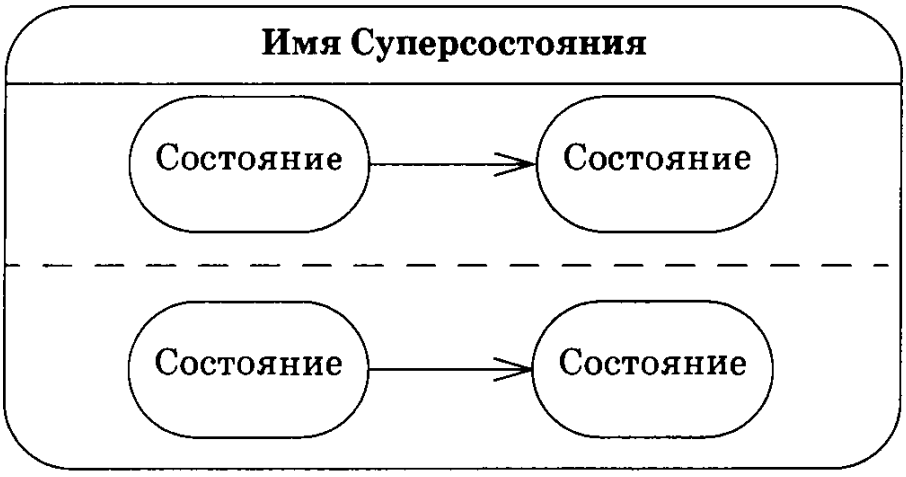


Диаграмма состояний



Параллельные состояния



В настоящее время широко распространенный унифицированный язык моделирования (UML) является официальным стандартом, рекомендованным консорциумом Object Management Group. UML представляет собой нотацию, которую должны знать и понимать все разработчики программного обеспечения. Возможности языка UML исключительно широки, но не все они важны в равной степени. Поэтому авторы первого издания «UML в кратком изложении», заслуженно признанного лучшим кратким руководством по основам языка UML, сочли возможным рассмотреть лишь самые важные его аспекты. Второе издание «UML. Основы», сохранив краткий стиль изложения, позволяет быстро изучить язык и приступить к его применению. Материал книги существенно обновлен и дополнен диаграммами вариантов использования, диаграммами деятельности и расширения кооперации, а также новым приложением, детально описывающим отличия разных версий языка UML.

Книга написана для тех, кто знаком с основами объектно-ориентированного анализа и проектирования. Она начинается с обзора истории возникновения, разработки и обоснования языка UML. Затем рассматривается возможность интеграции языка UML в объектно-ориентированный процесс разработки. М. Фаулер рассматривает в контексте UML различные методы моделирования, такие как варианты использования, диаграммы классов и диаграммы взаимодействия, описывает ясно и кратко нотацию и семантику. Уделено внимание полезным не-UML методам: CRC-картам и образцам. В книге приводятся практические рекомендации, основанные на 12-летнем опыте автора, а также небольшой пример программы на языке Java, иллюстрирующий реализацию UML-проекта. Освоив ключевые аспекты языка, читатели приобретут базовые знания, необходимые для построения моделей и углубленного изучения UML.

Мартин Фаулер – инициатор применения объектной технологии к информационным бизнес-системам. Последние двенадцать лет он является консультантом в области объектной технологии таких компаний, как Citibank, Chrysler Corporation, IBM, Andersen Consulting и Sterling Software. Им написана известная книга «Analysis Patterns: Reusable Object Models», а также «Refactoring: Improving the Design of Existing Code» – руководство для профессиональных программистов, заинтересованных в улучшении структуры существующего кода.

Кендалл Скотт – технический писатель, специализирующийся в области финансовых и бухгалтерских приложений, руководитель группы Software Documentation Wizards.

«Эта книга – большая удача. Она снабжена надежными советами, сформулированными в краткой и ясной форме. Прекрасно описана сущность нотации, но автор идет дальше этого, давая очень доступное понимание приложений методов языка UML».

– Jennifer Stapleton, Vice President Technical, British Computer Society

«UML в кратком изложении» показывает, как можно сказать много полезных вещей о компьютерных технологиях в столь небольшой книге».

– Gregory V. Wilson, Dr. Dobb's Journal

Категория: UML

Уровень подготовки читателей: Средний

ADDISON-WESLEY

Издательство «Символ-Плюс»
(812) 324-5353, (095) 945-8100

ISBN 5-93286-032-4

9 785932 860328

www.symbol.ru