UML Distilled

A Brief Guide to the Standard Object Modeling Language

Third Edition

Martin Fowler

Addison-Wesley

UML Основы

Краткое руководство
по стандартному языку
объектного моделирования

Третье издание

Мартин Фаулер

Санкт-Петербург
2005

Мартин Фаулер

UML. Основы, 3-е издание

Перевод А. Петухова

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*В. Шальнев*
Редактор	*В. Овчинников*
Корректура	*О. Макарова*
Верстка	*Н. Гриценко*

Фаулер М.
UML. Основы, 3-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2004. – 192 с., ил.
ISBN 5-93286-060-X

Третье издание бестселлера Фаулера «UML. Основы» охватывает UML 2 – версию, которая существенно отличается от всех предыдущих. Но основная формула успеха этой книги не претерпела изменений. До сих пор она, бесспорно, остается лучшим кратким и точным руководством по применению UML.

Главное достоинство книги заключается в кратком и сжатом изложении сути UML и особенностей применения этого языка в современном процессе разработки ПО. В книге описаны все главные типы диаграмм UML, рассказано, для чего они предназначены и какие нотации применяются при их создании и чтении. Это диаграммы классов, последовательности, объектов, пакетов, развертывания, прецедентов, состояний, деятельности, составных структур, компонентов, обзора взаимодействия, коммуникационные и временные.

Фаулер не только в ясной и доступной манере описывает ключевые аспекты языка UML, но и четко показывает ту роль, которую UML играет в процессе разработки. Замечательные примеры моделирования являются результатом многолетнего опыта работы автора в области проектирования и моделирования.

ISBN 5-93286-060-X
ISBN 0-321-19368-7 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 9.12.2004. Формат 70x100 1/16 . Печать офсетная.
Объем 12 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Посвящается Синди

Оглавление

Список иллюстраций

Отзывы

«Книга *UML Distilled* остается лучшим введением в нотации UML. Живой и прагматичный стиль Мартина прекрасно воспринимается, и я искренне рекомендую эту книгу».

– Крэйг Ларман (Craig Larman),
автор книги «Applying UML and Patterns»

«Фаулер пробивает путь сквозь сложности UML, помогая пользователям быстро познакомиться с UML».

– Джим Рамбо (Jim Rambaugh),
автор и один из создателей UML

«*UML Distilled* Мартина Фаулера – это прекрасный способ познакомиться с UML. Большинство пользователей найдут в этой книге все необходимое для успешного применения UML. С точки зрения Мартина, UML можно использовать различными путями, но наибольшее признание он получил как инструмент эскизного моделирования. Эта книга прекрасно выполняет работу по выявлению сущности UML. Настоятельно рекомендую».

– Стив Кук (Steve Cook), разработчик ПО, Microsoft

«Небольшие книги по UML лучше, чем большие. До сих пор эта книга остается лучшим кратким изданием по UML. Фактически это лучшая небольшая книга по многим темам».

– Алистер Кокборн (Alistair Cockburn),
автор и президент Humans and Technology

«Эта книга исключительно полезна, легко читается и – одно из главных ее достоинств – в восхитительно краткой манере охватывает значительное количество тем. Если вы собираетесь приобрести только одну книгу по UML, то должны купить именно эту».

– Энди Кармайкл (Andy Carmichael),
BetterSoftwareFaster, Ltd.

«Если вы используете UML, то эта книга всегда должна быть рядом».

– Джон Крупи (John Crupi), Sun Microsystems,
соавтор книги «Core J2EE Patterns»

«Все, кто занимается моделированием с применением UML, изучает UML, читает про UML или разрабатывает UML-инструменты, должны иметь последнее издание этой книги (у меня есть все издания). В ней много хорошей, полезной информации. В общем, информации достаточно, чтобы быть полезной, но не слишком много, чтобы стать скучной».

– Джон Керн (Jon Kern), разработчик моделей

«Это прекрасная отправная точка для изучения основ UML».

– Скотт В. Амблер (Scott W. Ambler),
автор книги «Agile Modeling»

«В высшей степени практичное описание языка UML и его применения, с достаточной степенью юмора, позволяющего удерживать внимание читателя. Воистину, «В плавательной метафоре больше нет воды».

– Стефан Меллор (Stephen J. Mellor),
соавтор книги «Executable UML»

«Это идеальная книга для тех, кто хочет использовать UML, но не желает читать толстые справочники по UML и исследовательские статьи. Мартин Фаулер отбирает все важные технологии, необходимые для использования UML при разработке эскизов, освобождая читателя от сложных и редко используемых возможностей UML. У читателей не будет недостатка в предложениях по дальнейшему изучению. Читатель получает советы, основанные на опыте. Это краткая и легко читаемая книга, посвященная основным аспектам UML и связанным с ними понятиями объектно-ориентированных технологий».

– Павел Хруби (Pavel Hruby),
Microsoft Business Solutions

«Подобно всем хорошим разработчикам программного обеспечения, Фаулер улучшает свой продукт с каждой итерацией. Это единственная книга, которой я пользуюсь, когда даю уроки UML, и которую рекомендую для изучения».

– Чарльз Ашбахер (Charles Ashbacher),
президент/CEO, Charles Ashbacher Technologies

«Должно быть больше книг, подобных *UML Distilled*, – кратких и легко читаемых. Мартин Фаулер выбирает разделы UML, которые вам нужны, и представляет их в удобной для чтения форме. Авторский опыт применения языка моделирования для документирования проекта имеет бульшую ценность, чем простое описание этой технологии».

– Роб Персер (Rob Purser), Purser Consulting, LLC.

Предисловие к третьему изданию

С древних времен большинству талантливых архитекторов и одаренных дизайнеров известен закон экономии. Независимо от формы, то ли в виде парадокса («чем меньше, тем больше»), то ли в виде коэна (разум дзэна – это разум новичка), он имеет непреходящее значение: Сократи все до его сути, так чтобы форма пребывала в гармонии с содержанием. Лучшие архитекторы и дизайнеры – от пирамид до Оперного театра в Сиднее, от построений Неймана до UNIX и языка Smalltalk – старались следовать этому универсальному и вечному правилу.

Я понимаю, что значит бриться Бритвой Оккама, поэтому когда я собираюсь вести разработку или читать, то ищу проекты и книги, в которых соблюдается закон экономии. Следовательно, я одобряю книгу, которую вы сейчас читаете.

Это мое замечание может удивить вас на первых порах. Я часто заглядывал в объемные и компактные спецификации, которые определяют UML (Unified Modeling Language – унифицированный язык моделирования). Эти спецификации позволяют инструментам поставщиков реализовывать UML, а методологам применять его. За семь лет мне довелось руководить большими международными командами по стандартизации, которые занимались спецификациями версий UML 1.1 и 2.0, а также нескольких менее важных промежуточных версий. В течение этого времени UML добавил в выразительности и точности, а также приобрел никому не нужную сложность как результат процесса стандартизации. Печально, что процесс стандартизации лучше известен компромиссами в области дизайна со стороны комитета, нежели своей расчетливой элегантностью.

Что может извлечь из книги Мартина, посвященной основам UML 2.0, специалист, хорошо знающий разные скрытые мелочи спецификации UML? Вполне достаточно, то же можете и вы. Мартин умело сократил большой и сложный язык до практичного подмножества, эффективность которого он доказал на практике. При подготовке нового издания своей книги он не пошел по легкому пути, который диктовала тактика простого добавления страниц. Когда язык разросся, Мартин попрежнему придерживался своей цели, ища «наиболее полезную со-

ставляющую языка UML» и рассказывая вам именно о ней. Составляющая, на которую он ссылается, – это те мистические 20% языка UML, которые помогают выполнять 80% работы. Поймать и приручить этого иллюзорного зверя – значительный успех!

Это тем более поразительно, что Мартин достигает своей цели, излагая материал в удивительной, притягательной разговорной манере. Донося до нас свою точку зрения и рассказывая при этом анекдоты, он делает свою книгу приятной для чтения и тем самым напоминает нам, что системы архитектуры и дизайна должны быть продуктивными и в то же время оригинальными. Если мы следуем коэну экономии до конца, то нужно признать, что моделирование проектов с помощью языка UML должно быть таким же приятным, какими были для нас уроки рисования и живописи в средней школе. UML должен стать громоотводом наших творческих способностей, а также лазером для выжигания четко определяющих систему планов, которые третья сторона могла бы запросить и построить по ним такую же систему. Последнее является серьезным испытанием для любого настоящего языка проектирования.

Для такой тонкой книжки это нетривиальная задача. Вы можете получить от изучения подхода Мартина к моделированию столь же много, как от его описания UML 2.0.

Мне было приятно работать с Мартином над улучшением подбора и точности возможностей языка UML 2.0, объясняемых в этой версии. Мы должны иметь в виду, что все современные языки, как естественные, так и искусственные, должны развиваться или исчезнуть. Выбор Мартином новых свойств языка в соответствии с его предпочтениями и в соответствии с предпочтениями других профессионалов – это решающий момент процесса пересмотра UML. Они поддерживают язык в жизнеспособном состоянии и помогают ему эволюционировать в процессе естественного отбора на рынке.

Значительное количество многообещающих работ остаются вне управляемого моделями способа разработки, который становится ведущим, но меня поддерживают книги, подобные этой, которые понятно объясняют основы моделирования на языке UML и показывают его использование на практике. Я надеюсь, что вы, как и я, с ее помощью получите новые знания и используете приобретенные вами навыки для повышения качества моделирования программного обеспечения.

Крис Кобрин (Cris Kobryn)
Chair, U2 Partners' UML 2.0 Submission Team
Chief Technologist, Telelogic

Предисловие к первому изданию

Когда мы приступили к созданию унифицированного языка моделирования (Unified Modeling Language, UML), 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 полагали, 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.

Само 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 последователи помогли довести ее до успешного завершения, и мы благодарны им за их 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 другой, ближе к решению поставленной задачи.

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

На протяжении жизни мне много раз улыбалась удача; одним из больших подарков фортуны было то, что я оказался в нужном месте, вооруженный необходимыми знаниями, в результате чего в 1997 году было написано первое издание этой книги. В то время в хаотическом мире объектно-ориентированного (ОО) моделирования только начинался процесс объединения под эгидой унифицированного языка моделирования (Unified Modeling Language, UML). С тех пор UML стал стандартом графического моделирования не только объектов, но и программного обеспечения в целом. Мне повезло, что эта книга была самой популярной по языку UML, разойдясь тиражом более четверти миллиона экземпляров.

Конечно, мне это очень приятно, но зачем вам покупать мою книгу?

Я люблю подчеркивать, что это тонкая книжка. Ее цель не в том, чтобы детально осветить каждый аспект языка UML, растущего с каждым годом. Я стремлюсь найти наиболее полезную часть языка и рассказать вам именно о ней. Хотя более объемная книга дает более детальное описание, но и читать ее нужно дольше. А время – самый ценный капитал, который вы вкладываете при чтении книги. Сохраняя небольшой размер книги, я сэкономил вам время, выбирая самое лучшее, и избавил вас от необходимости самостоятельно выполнить эту работу. (К сожалению, «меньше» не означает пропорционально дешевле; существует определенная фиксированная стоимость издания высококачественной технической книги.)

Один из мотивов, побуждающих приобрести данную книгу, – желание получить начальные сведения по UML. Это маленькая книжка, и с ее помощью можно быстро постичь основы языка. Что касается профессионального освоения, то более подробную информацию можно найти в книге «User Guide» [6] или «Reference Manual» [40].

Эта книга может также служить удобным справочником по наиболее общим разделам языка UML. В ней есть не все, зато она много легче большинства других книг по UML и ее можно носить с собой повсюду.

Это книга с ярко выраженным мнением. Я долгое время имел дело с объектами и точно знаю, что работает, а что – нет. Любая книга отражает мнение автора, и я также не стараюсь скрывать свое. Поэтому ес-

ли вы ищете нечто, имеющее оттенок объективности, то, возможно, вам потребуется еще что-нибудь.

Многие разработчики говорили мне, что эта книга является хорошим введением в объекты, однако при ее написании у меня не было такой мысли. Если вам требуется введение в ОО, я бы рекомендовал книгу К. Лармана (Craig Larman) [29].

Многие разработчики, интересующиеся UML, используют некоторый инструментарий. Эта книга посвящена стандартному применению UML в рамках принятых соглашений, и в ней не рассматриваются подробности поддержки языка различными инструментами. Несмотря на то что UML справился с Вавилонской башней нотаций, существовавших до UML, осталось множество досадных различий между тем, что показывают инструменты, и тем, что они позволяют делать в процессе рисования UML-диаграмм.

Я не рассказываю в этой книге об MDA (Model Driven Architecture) – архитектуре, основанной на модели. Распространено мнение, что это одно и то же, но многие разработчики применяют язык UML, совершенно не интересуясь MDA. Тем, кто хочет узнать об MDA больше, я бы посоветовал начать с данной книги, чтобы сначала познакомиться с UML, а затем перейти к книге, посвященной MDA.

Хотя главной темой этой книги является язык UML, я включил в нее дополнительный материал о приемах, которые очень полезны в объектно-ориентированном проектировании, например, приведена информация о CRC-карточках. UML – это только часть того, что необходимо знать для успешной работы с объектами, и я думаю, что он играет важную роль при подготовке к освоению других приемов.

В такой небольшой книге, как эта, невозможно детально объяснить, как UML соотносится с исходным кодом, в частности потому, что не существует стандартного способа проведения такого соответствия. Однако я демонстрирую некоторые приемы программирования для реализации элементов UML. Мои примеры написаны на Java и C#, поскольку я обнаружил, что они понятны более широкой аудитории. Не надо думать, что они мне больше нравятся. Для этого я слишком много написал на Smalltalk!

Почему нужно заниматься UML?

Нотации визуального проектирования применяются уже довольно долго. На мой взгляд, они играют основную роль для взаимопонимания. Хорошая диаграмма часто помогает обмениваться идеями о проекте, особенно когда вы хотите избежать излишне подробного объяснения. Диаграммы также помогают понять и программную систему, и бизнес-план. Когда группа разработчиков пытается в чем-то разобраться, диаграммы помогают установлению взаимопонимания и распространению такого понимания в команде. Диаграммы, по крайней

мере пока, не заменяют текстовые языки программирования, но способны оказать существенную помощь.

Многие уверены, что в будущем приемы визуального моделирования выйдут на ведущие роли при создании программного обеспечения. Я отношусь к этому более скептически, но определенно полезно понять, что можно сделать с помощью этих нотаций, а что нельзя. Наряду с графическими элементами значимость UML основана на его широком распространении и стандартизации в рамках сообщества разработчиков, применяющих объектно-ориентированные технологии. Язык UML стал не только доминирующим визуальным инструментом в мире объектно-ориентированного моделирования, но также получил признание и за его пределами.

Структура книги

Глава 1 представляет собой введение в UML: в ней описывается собственно язык, рассказано, что он означает для разных разработчиков и откуда он появился.

В главе 2 обсуждается процесс создания программного обеспечения. Хотя это совершенно не зависит от UML, я считаю, что необходимо понять этот процесс, чтобы увидеть контекст, подобный UML. В частности, важно оценить роль итеративной разработки, лежащей в основе подхода к процессу в большинстве ОО-сообществ.

В оставшейся части книги рассмотрены диаграммы UML различных типов. Главы 3 и 4 посвящены двум наиболее полезным разделам UML – диаграммам классов (основная часть) и диаграммам последовательностей. Это тонкая книжка, но я уверен, что приемы, о которых я рассказываю в этих главах, позволят вам оценить значимость этого языка. UML велик и продолжает расти, но весь UML вам не потребуется.

В главе 5 подробно рассмотрены менее важные, но все же полезные элементы диаграмм классов. В главах с 6 по 8 описываются три полезные диаграммы, которые еще более проясняют *структуру* системы: диаграммы объектов, диаграммы пакетов и диаграммы развертывания.

В главах с 9 по 11 рассматриваются другие полезные поведенческие приемы: прецеденты, диаграммы состояний (хотя официально они известны как диаграммы конечных автоматов, чаще всего их называют диаграммами состояний) и диаграммы деятельности. Главы с 12 по 17 очень короткие и посвящены диаграммам, в большинстве случаев имеющим менее важное значение, поэтому для них я привел небольшие примеры и краткие объяснения.

Изложение включает обзор наиболее полезных элементов каждой нотации. Я часто слышал от читателей, что это наиболее ценная часть книги. Возможно, вы найдете удобным обращаться к ним во время чтения других разделов книги.

Изменения в третьем издании

Обладатели более раннего издания этой книги, возможно, зададутся вопросом об отличиях или, что важнее, о необходимости приобрести новое издание.

Главным толчком к написанию третьего издания было появление UML 2. В него было добавлено множество новых элементов, в том числе несколько новых типов диаграмм. Даже в знакомых диаграммах применяется много новых нотаций, таких как фреймы взаимодействия в диаграммах последовательностей. Тем, кто хочет быть в курсе происходящего, но не желает утомлять себя чтением спецификации (я определенно не рекомендовал бы этого делать!), эта книга может предложить хороший обзор.

Кроме того, я воспользовался этой возможностью, чтобы полностью переписать бтльшую часть книги, обновив примеры и текст многим из того, что я изучил, преподавая и применяя UML в течение последних пяти лет. Поэтому, несмотря на то что дух этой сверхтонкой книги остался нетронутым, большинство слов в ней новые.

Все эти годы я усердно работал, пытаясь по мере сил сохранить актуальность материала. Пока UML изменялся, я изо всех сил старался не отстать от него. В основе этой книги лежит проект UML 2, который был принят соответствующим комитетом в июне 2003 года. Маловероятно, что между этим голосованием и другими, более формальными голосованиями произойдут дальнейшие изменения, поэтому я чувствую, что UML 2 теперь достаточно стабилен, чтобы отдать книгу в печать. Я буду размещать информацию об обновлениях на веб-сайте *http://martinfowler.com*.

Благодарности

Долгие годы усилия многих людей составляли успех этой книги. Первыми хочу поблагодарить Картера Шанклина (Carter Shanklin) и Кендалла Скотта (Kendall Scott). Картер был тем самым редактором издательства Addison-Wesley, кто предложил мне написать эту книгу. Кендалл Скотт помогал мне объединять первые два издания, работая над текстом и графикой. Вместе они справились с чрезвычайно трудной задачей подготовки первого издания в исключительно короткие сроки, сохраняя то высокое качество, которое читатели ожидают от издательства Addison-Wesley. Они также отслеживали изменения в первое время существования языка UML, когда все казалось нестабильным.

Джим Оделл был моим наставником и гидом в начале моей карьеры. Он также глубоко вникал в технические и личные споры упрямых методологов, высказывавших свое несогласие или сходство во взглядах во время становления единого стандарта. Его основательный вклад в эту

книгу трудно измерить, и я готов поспорить, что его вклад в UML не меньше.

UML – это порождение стандартов, а у меня на стандарты аллергия. Поэтому, чтобы знать, как идут дела, мне необходимо иметь сеть шпионов, которые держали бы меня в курсе всех происков комитетов. Без этих шпионов, включая Конрада Бока (Conrad Bock), Стива Кука (Steve Cook), Криса Кобрина (Cris Kobryn), Джима Одела (Jim Odell), Гуса Ремакерса (Guus Ramackers) и Джима Рамбо (Jim Rumbaugh), я оказался бы в затруднительном положении. Все они давали мне полезные советы и отвечали на глупые вопросы.

Гради Буч (Grady Booch), Айвар Джекобсон (Ivar Jacobson) и Джим Рамбо (Jim Rumbaugh) известны как «трое друзей». Они много лет не обращали внимания на мои выходки, поддерживали и ободряли меня во время работы над книгой. Помните, что мои колкости обычно вырастают из нежной признательности.

Ключ к качеству книги – рецензенты, и от Картера я узнал, что их никогда не бывает слишком много. Рецензентами предыдущих изданий этой книги были Симми Коччар Баргава (Simmi Kochhar Bhargava), Гради Буч (Grady Booch), Эрик Эванс (Eric Evans), Том Хэдфилд (Tom Hadfield), Айвар Джекобсон (Ivar Jacobson), Рональд Джеффрис (Ronald E. Jeffries), Джошуа Кериевски (Joshua Kerievsky), Хелен Клейн (Helen Klein), Джим Оделл (Jim Odell), Джим Рамбо (Jim Rumbaugh) и Вивек Салгар (Vivek Salgar).

У третьего издания также были прекрасные рецензенты:

Конрад Бок (Conrad Bock)
Энди Кармайкл (Andy Carmichael)
Алистер Кокбурн (Alistair Cockburn)
Стив Кук (Steve Cook)
Люк Гохман (Luke Hohmann)
Павел Хруби (Pavel Hruby)
Джон Керн (Jon Kern)
Крис Кобрин (Cris Kobryn)
Крейг Ларман (Craig Larman)
Стив Меллор (Steve Mellor)
Джим Оделл (Jim Odell)
Алан О'Каллахан (Alan O'Callaghan)
Гус Рамакерс (Guus Ramackers)
Джим Рамбо (Jim Rumbaugh)
Тим Зельтцер (Tim Seltzer)

Все рецензенты потратили время на чтение рукописи, и каждый из них нашел по крайней мере по одной постыдной грубой ошибке. Мои искренние благодарности им всем. Все оставшиеся грубые ошибки целиком на моей совести. Когда я их обнаружу, я помещу список исправлений в разделе книг сайта *martinfowler.com*.

В основной состав команды, разработавшей и написавшей спецификацию языка UML, входят Дон Бейсли (Don Baisley), Морган Бьеркандер (Morgan Bjцrkander), Конрад Бок (Conrad Bock), Стив Кук (Steve Cook), Филипп Десфрай (Philippe Desfray), Натан Дикман (Nathan Dykman), Андерс Ек (Anders Ek), Дэвид Франкел (David Frankel), Еран Гери (Eran Gery), Ойстен Хаген (Шystein Haugen), Шридхар Йенгар (Sridhar Iyengar), Крис Кобрин (Cris Kobryn), Биргер Меллер-Педерсен (Birger Mшller-Pedersen), Джеймс Оделл (James Odell), Гуннар Овергард (Gunnar Цvergaard), Карин Палмквист (Karin Palmkvist), Гус Рамакерс (Guus Ramackers), Джим Рамбо (Jim Rumbaugh), Бран Селик (Bran Selic), Томас Вейгерт (Thomas Weigert) и Ларри Вильямс (Larry Williams). Без них я бы вряд ли что-нибудь написал.

Павел Хруби (Pavel Hruby) разработал несколько прекрасных шаблонов для Visio, которые я использовал во многих диаграммах UML; их можно найти на *http://phruby.com*.

Многие обращались ко мне по Сети и лично с предложениями и вопросами и с указаниями на ошибки. Я не смог ответить всем вам, но мои благодарности не менее искренни.

Сотрудники моего любимого книжного магазина SoftPro в Верлингтоне, штат Массачусетс, предоставили возможность наблюдать за их складом, что позволило мне узнать, как на практике люди используют UML. Они угощали меня хорошим кофе, пока я был у них в гостях.

Ведущим редактором третьего издания был Майк Хендриксон (Mike Hendrickson). Ким Арни Малкахи (Kim Arney Mulcahy) руководил проектом, а также делал планировку и подчистку диаграмм. Джон Фуллер из издательства Addison-Wesley был выпускающим редактором, а Эвелин Пиле (Evelyn Pyle) и Ребекка Райдер (Rebecca Rider) помогали в редактировании и чтении корректуры книги. Я благодарю их всех.

Синди оставалась со мной все время, пока я упорно писал книгу. А потом закапывала полученный гонорар в саду. Мои родители дали мне хорошее образование – источник всего остального.

Мартин Фаулер
Мелроуз, Массачусетс
http://martinfowler.com

1

Введение

Что такое UML?

Унифицированный язык моделирования (UML) – это семейство графических нотаций, в основе которого лежит единая метамодель. Он помогает в описании и проектировании программных систем, в особенности систем, построенных с использованием объектно-ориентированных (ОО) технологий. Это определение в чем-то упрощенное. В действительности разные люди могут видеть в UML разные вещи. Это является следствием как собственной истории развития языка, так и различных точек зрения специалистов на то, что делает процесс разработки программного обеспечения эффективным. Поэтому моя задача в этой главе во многом заключается в построении общей картины книги и в объяснении различного видения и разнообразных способов применения UML разработчиками.

Графические языки моделирования уже продолжительное время широко используются в программной индустрии. Основная причина их появления состоит в том, что языки программирования не обеспечивают нужный уровень абстракции, способный облегчить процесс проектирования.

Несмотря на то что графические языки моделирования существуют уже достаточно давно, в среде разработчиков программного обеспечения очень много спорят об их роли. Эти споры оказывают непосредственное влияние на восприятие разработчиками самого языка UML.

UML представляет собой относительно открытый стандарт, находящийся под управлением группы OMG (Object Management Group – группа управления объектами), открытого консорциума компаний. Группа OMG была сформирована для создания стандартов, поддерживающих межсистемное взаимодействие, в частности взаимодействие объектно-ориентированных систем. Возможно, группа OMG более известна по стандартам CORBA (Common Object Request Broker Architecture – общая архитектура посредников запросов к объектам).

UML появился в результате процесса унификации множества объектно-ориентированных языков графического моделирования, процветавших в конце 80-х и в начале 90-х годов. Появившись в 1997 году, он отправил эту Вавилонскую башню в вечность, за что я и многие другие разработчики испытываем по отношению к нему глубокую благодарность.

Способы применения UML

Основу роли UML в разработке программного обеспечения составляют разнообразные способы использования языка, те различия, которые были перенесены из других языков графического моделирования. Эти отличия вызывают долгие и трудные дискуссии о том, как следует применять UML.

Чтобы разрешить эту сложную ситуацию, Стив Меллор (Steve Mellor) и я независимо пришли к определению трех режимов использования UML разработчиками: режим эскиза, режим проектирования и режим языка программирования. Безусловно, самый главный из трех, по крайней мере, на мой пристрастный взгляд, – это режим использования *UML для эскизирования*. В этом режиме разработчики используют UML для обмена информацией о различных аспектах системы. В режиме проектирования можно использовать эскизы при прямой и обратной разработке. При *прямой разработке* (*forward-engineering*) диаграммы рисуются до написания кода, а при *обратной разработке* (*reverse-engineering*) диаграммы строятся на основании исходного кода, чтобы лучше понять его.

Сущность эскизирования, или эскизного моделирования, в избирательности. В процессе прямой разработки вы делаете наброски отдельных элементов программы, которую собираетесь написать, и обычно обсуждаете их с некоторыми разработчиками из вашей команды. При этом с помощью эскизов вы хотите облегчить обмен идеями и вариантами того, что вы собираетесь делать. Вы обсуждаете не всю программу, над которой намереваетесь работать, а только самые важные ее моменты, которые вы хотите донести до коллег в первую очередь, или разделы проекта, которые вы хотите визуализировать до начала программирования. Такие совещания могут быть очень короткими: 10-минутное совещание по нескольким часам программирования или однодневное совещание, посвященное обсуждению двухнедельной итерации.

При обратной разработке вы используете эскизы, чтобы объяснить, как работает некоторая часть системы. Вы показываете не все классы, а только те, которые представляют интерес и которые стоит обсудить перед тем, как погрузиться в код. Поскольку эскизирование носит неформальный и динамичный характер и вам нужно делать это быстро и совместно, то наилучшим средством отображения является доска. Эскизы полезны также и в документации, при этом главную роль играет процесс передачи информации, а не полнота. Инструментами эскизного моделирования служат облегченные средства рисования, и

часто разработчики не очень придерживаются всех строгих правил UML. Большинство диаграмм UML, показанных в этой книге, как и в других моих книгах, представляют собой эскизы. Их сила в избирательности передачи информации, а не в полноте описания.

Напротив, язык *UML как* средство *проектирования* нацелен на полноту. В процессе прямой разработки идея состоит в том, что проект разрабатывается дизайнером, чья работа заключается в построении детальной модели для программиста, который будет выполнять кодирование. Такая модель должна быть достаточно полной в части заложенных проектных решений, а программист должен иметь возможность следовать им прямо и не особо задумываясь. Дизайнером модели может быть тот же самый программист, но, как правило, в качестве дизайнера выступает старший программист, который разрабатывает модели для команды программистов. Причина такого подхода лежит в аналогии с другими видами инженерной деятельности, когда профессиональные инженеры создают чертежи, которые затем передаются строительным компаниям.

Проектирование может быть использовано для всех деталей системы либо дизайнер может нарисовать модель какой-то конкретной части. Общий подход состоит в том, чтобы дизайнер разработал модели проектного уровня в виде интерфейсов подсистем, а затем дал возможность разработчикам поработать над реализацией подробностей.

При обратной разработке цель моделей состоит в представлении подробной информации о программе или в виде бумажных документов, или в виде, пригодном для интерактивного просмотра с помощью графического броузера. В такой модели можно показать все детали класса в графическом виде, который разработчикам проще понять.

При разработке моделей требуется более сложный инструментарий, чем при составлении эскизов, так как необходимо поддерживать детальность, соответствующую требованиям поставленной задачи. Специализированные CASE-средства (computer-aided software engineering – автоматизированная разработка программного обеспечения) попадают в эту категорию, хотя сам этот термин стал почти ругательным, и поставщики стараются его избегать. Инструменты прямой разработки поддерживают рисование диаграмм и копирование их в репозиторий с целью сохранения информации. Инструменты обратного проектирования читают исходный код, записывают его интерпретацию в репозиторий и генерируют диаграммы. Инструменты, позволяющие выполнять как прямую, так и обратную разработку, называются *двухсторонними* (*round-trip*).

Некоторые средства используют исходный код в качестве репозитория, а диаграммы используют его для графического представления. Такие инструменты более тесно связаны с программированием и часто встраиваются прямо в средства редактирования исходного кода. Мне нравится называть их «тройными» инструментами.

Граница между моделями и эскизами довольно размыта, но я думаю, что различия остаются в том, что эскизы сознательно выполняются неполными, подчеркивая важную информацию, в то время как модели нацелены на полноту, часто имея целью свести программирование к простым и до некоторой степени механическим действиям. Короче говоря, я бы определил эскизы как пробные элементы, а модели – как окончательные.

Чем дольше вы работаете с UML, а программирование становится все более механическим, тем очевиднее становится необходимость перехода к автоматизированному созданию программ. Действительно многие CASE-средства так или иначе генерируют код, что позволяет автоматизировать построение значительной части системы. В конце концов, вы достигнете такой точки, когда сможете описать с помощью UML всю систему и перейдете в режим использования *UML в качестве языка программирования*. В такой среде разработчики рисуют диаграммы, которые компилируются прямо в исполняемый код, а UML становится исходным кодом. Очевидно, что такое применение UML требует особенно сложных инструментов. (Кроме того, нотации прямой и обратной разработки теряют всякий смысл, поскольку UML и исходный код становятся одним и тем же.)

Один из интересных вопросов, касающихся UML как языка программирования, – это вопрос о моделировании логики поведения. UML 2 предлагает три способа моделирования поведения: диаграммы взаимодействия, диаграммы состояний и диаграммы деятельности. Все они имеют своих сторонников в сфере программирования. Если UML добьется популярности как язык программирования, то будет интересно посмотреть, какой из этих способов будет иметь успех.

Другая точка зрения разработчиков на UML находится где-то между его применением для концептуального моделирования и его применением для моделирования программного обеспечения. Большинство разработчиков используют UML для моделирования программного обеспечения. С *точки зрения программного обеспечения* элементы UML практически непосредственно отображаются в элементы программной системы. Как мы увидим впоследствии, отображение отнюдь не означает следование инструкциям, но когда мы используем UML, мы говорим об элементах программного обеспечения.

С *концептуальной точки зрения* UML представляет описание концепций предметной области. Здесь мы не столько говорим об элементах программного обеспечения, сколько занимаемся созданием словаря для обсуждения конкретной предметной области.

Нет строгих правил выбора точки зрения. Поскольку проблему можно рассматривать под разными углами зрения, то и способов применения существует довольно много. Некоторые инструменты автоматически преобразуют исходный код в диаграммы, трактуя UML как альтернативный вид исходного кода. Это в большей степени программный ракурс. Если же диаграммы UML применяются для того, чтобы проверить

Архитектура, управляемая моделью, и исполняемый UML

Когда говорят о UML, часто упоминают об *MDA* (Model Driven Architecture – архитектура, управляемая моделью) [27]. По сути дела, MDA представляет собой стандартный подход к использованию UML в качестве языка программирования; этот стандарт находится под управлением группы OMG, как и сам UML. Создавая систему моделирования, соответствующую MDA, поставщики могут разработать модели, способные работать и в MDA-совместимом окружении.

Говоря об MDA, часто подразумевают и UML, поскольку MDA использует UML в качестве основного языка моделирования. Но, конечно, вы не обязаны следовать MDA, применяя UML.

MDA разделяет процесс разработки на две основные части. Разработчики моделей представляют конкретное приложение, создавая *PIM* (Platform Independent Model – модель, не зависящая от платформы). PIM – это модель UML, не зависящая от какой-то конкретной технологии. Затем инструменты могут превратить PIM в PSM (Platform Specific Model – модель, зависящая от платформы). PSM – это модель системы, нацеленная на определенную среду выполнения. Другие инструменты берут PSM и генерируют код для этой платформы. В качестве PSM можно использовать UML, но это необязательно.

Поэтому если вы хотите создать систему складского учета с использованием MDA, вам придется начать с единой модели PIM вашей системы. Затем при желании запустить эту систему складского учета в J2EE или .NET вы должны будете использовать инструменты каких-либо производителей для создания двух моделей PSM – по одной на каждую из этих двух платформ.

Если процесс перехода от модели PIM к модели PSM и окончательной программе полностью автоматизирован, то мы используем UML в качестве языка программирования. Если на каком-нибудь этапе присутствует ручная обработка, то мы используем UML в режиме проектирования.

Стив Меллор (Steve Mellor) долгое время активно работал в этой области и с недавнего времени стал употреблять термин *исполняемый UML* (Executable UML) [32]. Исполняемый UML похож на MDA, но использует немного другую терминологию. Точно так же вы начинаете с модели, не зависящей от платформы, которая эквивалентна MDA-модели PIM. Однако на следующем этапе применяется компилятор модели (Model Compiler), для того чтобы за один прием превратить UML-модель в готовую к развертыванию систему; поэтому модель PSM не нужна. Как и предполагает термин *компилятор*, этот этап полностью автоматизирован.

Компиляторы модели основаны на повторно используемых прототипах. *Прототип* (*archetype*) описывает способ превращения исполняемого UML в соответствующую программную платформу. Поэтому в случае с системой складского учета придется купить компилятор модели и два прототипа (J2EE и .NET). Примените эти прототипы к вашему исполняемому UML, и у вас будет две версии системы складского учета.

Исполняемый UML не использует полный стандарт UML; многие конструкции языка считаются ненужными и не применяются. Поэтому исполняемый UML проще, чем полный.

Все это звучит хорошо, но насколько это реалистично? На мой взгляд, здесь есть два аспекта. Во-первых, вопрос об инструментах: достаточно ли они развиты, чтобы выполнить поставленную задачу. Этот фактор со временем меняется; определенно, как я уже писал, они не слишком широко применяются, и я не многие из них видел в действии.

Более фундаментальным аспектом является сама идея применения UML в качестве языка программирования. С моей точки зрения, использовать UML как язык программирования стоит, только если в результате получается нечто более продуктивное, чем в случае применения другого языка программирования. Однако исходя из своего опыта работы в различных графических средах разработки, я не стал бы это утверждать. Даже если UML и более продуктивен, надо еще накопить критическую массу пользователей, чтобы принять его в качестве основного направления. UML сам по себе представляет большое препятствие. Подобно многим пользователям, имеющим опыт работы с языком Smalltalk, я считаю его более продуктивным, чем многие современные основные языки. Но в настоящее время Smalltalk представляет только небольшую нишу в пространстве языков, и я не наблюдаю большого количества проектов, написанных на нем. Чтобы избежать судьбы языка Smalltalk, UML должен быть счастливчиком, даже если он самый лучший.

и понять различные значения терминов «пул активов» (asset pool) с группой бухгалтеров, то следует принять точку зрения значительно более близкую к концептуальной.

В предыдущем издании этой книги я разделил программную точку зрения на спецификацию (specification) и реализацию (implementation). Сейчас я понял, что на практике довольно трудно провести между ними точную границу, поэтому чувствую, что не нужно больше беспокоиться о различиях между ними. Однако я всегда был склонен в своих диаграммах делать ударение на интерфейсе, а не на реализации.

Следствием различных способов применения UML является масса споров о том, что означают диаграммы UML и как они связаны с осталь-

ным миром. Особенно это влияет на отношение между UML и исходным кодом. Некоторые разработчики считают, что UML нужно применять для создания модели, не зависящей от языка программирования, который используется для реализации проекта. Другие убеждены в том, что модель, не зависимая от языка, – это оксюморон с выраженным ударением на слово «морон» (moron – идиот).

Другое различие во взглядах относится к вопросу о сущности UML. По-моему, большинство пользователей UML, особенно создателей эскизов, видят сущность UML в его диаграммах. Однако авторы UML считают диаграммы вторичным, а первичным признают метамодель UML. Диаграммы же – лишь представление метамодели. Такая точка зрения имеет смысл также для разработчиков, использующих UML в режиме проектирования и в режиме языка программирования.

Итак, всякий раз когда вы читаете что-нибудь, относящееся к UML, важно понимать точку зрения автора. Только тогда вы сможете правильно оценить эти часто горячие дискуссии, которые вызывает UML.

Написав это, я должен пояснить свои пристрастия. Практически вся моя работа с UML посвящена созданию эскизов. Я считаю, что эскизы UML полезны и в процессе прямой, и в процессе обратной разработки, а также и с концептуальной точки зрения, и в программном ракурсе.

Я не любитель детальных моделей, созданных в процессе прямого проектирования, поскольку убежден, что они слишком трудно реализуются и замедляют процесс разработки. Создание моделей уровня интерфейсов подсистем вполне разумно, но даже в этом случае вы должны быть готовы к изменению этих интерфейсов, когда разработчики будут реализовывать взаимодействие интерфейсов. Значение моделей в процессе обратной разработки зависит от того, как работает инструментарий. Если он применяется в качестве динамического броузера, то он может быть очень полезным; если же он генерирует большой документ, то вся его работа сводится к пустому переводу бумаги.

Я считаю, что применение UML в качестве языка программирования – удачная идея, но сомневаюсь, что он когда-нибудь будет широко использоваться в этом качестве. Я не уверен, что для большинства программных задач графическое представление более продуктивно, чем текстовое, и даже если это так, то вряд ли такой язык получит широкое распространение.

В соответствии с моими пристрастиями эта книга посвящена, главным образом, использованию UML для создания эскизов. К счастью, это имеет смысл при написании краткого руководства. Я не в состоянии показать все возможности других режимов работы UML в книге такого объема, но эта небольшая книга является хорошим введением в другие книги, которые могут решить эту задачу. Поэтому если вас интересуют другие режимы использования UML, я предлагаю считать эту книгу введением и обратиться к другим книгам, когда это будет необ-

ходимо. Для тех же, кого интересуют только эскизы, данное издание может оказаться именно тем, что нужно.

Как мы пришли к UML

Надо признать, что я люблю историю. Мое любимое легкое чтение – это хорошая историческая книга. Но я понимаю, что это нравится не всем. Я говорю здесь об истории, поскольку считаю, что во многих отношениях трудно оценить место, занимаемое UML, не зная истории его развития.

В 80-х годах объекты начали выходить из исследовательских лабораторий и делать свои первые шаги в направлении «реального» мира. Язык Smalltalk был реализован на платформе, пригодной для практического использования; появился на свет и C++. В то же время разработчики начали задумываться об объектно-ориентированных языках графического моделирования.

Основные книги по объектно-ориентированным языкам графического моделирования появились между 1988 и 1992 годами. Ведущими фигурами в этом деле были Гради Буч (Grady Booch) [5]; Питер Коуд (Peter Coad) [7], [8]; Айвар Джекобсон (Ivar Jacobson) (Objectory) [24]; Джим Оделл (Jim Odell) [34]; Джим Рамбо (Jim Rumbaugh) (OMT) [38], [39]; Салли Шлаер (Sally Shlaer) и Стив Меллор (Steve Mellor) [41], [42]; Ребекка Вирфс-Брок (Rebecca Wirfs-Brock) (Responsibility Driven Design) [44].

Каждый из перечисленных авторов в то время был неформальным лидером группы профессионалов, приверженцев этих идей. Все эти методы были очень похожи, хотя между ними и существовали небольшие, но нередко раздражающие отличия. Идентичные базовые концепции обязательно проявлялись в самых разнообразных нотациях, которые вызывали путаницу в головах моих клиентов.

В это горячее время о стандартизации говорили в той же степени, в какой ее игнорировали. Команда из группы OMG пыталась разбираться со стандартизацией, но в результате получала только письма с протестами от всех ведущих методологов. (Есть такой старый анекдот. Вопрос: В чем разница между методологом и террористом? Ответ: С террористом можно вести переговоры.)

Катастрофическим событием, инициировавшим появление UML, стали уход Джима Рамбо из GE и его встреча с Гради Бучем в Rational (теперь это подразделение IBM). С самого начала было ясно, что союз Буч/Рамбо может создать критическую массу[1] этого бизнеса. Гради и Джим провозгласили, что «война методов завершена – мы победили»,

[1] Критическая масса (в маркетинге) – обязательный набор новшеств, которые должны быть присущи товару, чтобы он считался современным. – *Примеч. ред.*

по существу заявляя, что они собираются достигнуть стандартизации, пойдя по «пути Microsoft». Некоторые методологи проектирования предложили создать анти-Бучевскую коалицию.

К конференции OOPSLA '95 Гради и Джим подготовили свое первое публичное описание их совместного метода – версию 0.8 документации по *Унифицированному методу* (*Unified Method*). И что более существенно, они объявили, что фирма Rational Software купила Objectory и что Айвар Джекобсон должен присоединиться к Унифицированной команде. Rational устроила шикарный прием, празднуя выход версии 0.8. (Гвоздем программы стало первое публичное выступление поющего Джима Рамбо; мы все надеемся, что оно также было и последним.)

На следующий год процесс стал более открытым. Группа OMG, которая раньше в основном стояла в стороне, теперь стала играть активную роль. Rational вынуждена была принять идеи Айвара, а также работать с другими партнерами. Более существенно то, что группа OMG решила взять на себя ведущую роль.

Важно понять, почему вступила в дело группа OMG. Методологи проектирования, как и авторы книг, любят думать, что они важные персоны. Но я не думаю, что вопли авторов книг могут быть хотя бы услышаны группой OMG. Причиной участия группы OMG стали крики поставщиков программных средств. Поставщики испугались, что стандарт, контролируемый фирмой Rational, даст продуктам фирмы неоправданное конкурентное преимущество. В результате производители побудили группу OMG к действиям под флагом необходимости взаимодействия CASE-систем. Это был очень важный аргумент, поскольку для группы OMG взаимодействие значило очень много. Появилась идея создать язык UML, который бы позволил CASE-средствам свободно обмениваться моделями.

Мэри Лумис (Mary Loomis) и Джим Оделл возглавили инициативную группу, созданную для решения поставленной задачи. Оделл дал ясно понять, что готов предоставить свой метод для стандарта, но не желает поддерживать стандарт, продиктованный фирмой Rational. В январе 1997 года различные организации представили на рассмотрение свои предложения по стандартизации методов, которые бы способствовали обмену моделями. Фирма Rational сотрудничала с несколькими организациями и в соответствии со своим планом представила версию 1.0 документации по UML – первое создание, соответствующее имени Унифицированный язык моделирования (Unified Modeling Language).

Затем последовал короткий период выкручивания рук, когда объединялись различные предложения. Группа OMG приняла окончательную версию 1.1 в качестве официального стандарта OMG. Позднее были созданы другие версии. Версия 1.2 была целиком косметическая, версия 1.3 – более значимой. В версию 1.4 было введено множество детализированных понятий о компонентах и профилях, а в версию 1.5 добавили семантику действия.

Когда специалисты говорят о UML, они называют его создателями главным образом Гради Буча, Айвара Джекобсона и Джима Рамбо. Чаще всего их называют «Трое друзей» (Three Amigos), хотя шутники любят опускать первый слог второго слова. Несмотря на то что они заработали свое доброе имя в основном в связи с UML, я думаю, что не совсем справедливо отдавать им всю славу. Нотация UML впервые была сформирована в Унифицированном методе Буча/Рамбо. С тех пор была проведена большая работа в комитетах группы OMG. На этой стадии Джим Рамбо был лишь одним из тройки, которая выполнила свои тяжелые обязательства. Я считаю, что основную благодарность за создание UML заслужили именно члены комитета UML.

Нотации и метамодели

Язык UML в своем нынешнем состоянии определяет нотацию и метамодель. **Нотация** представляет собой совокупность графических элементов, которые применяются в моделях; она и есть синтаксис данного языка моделирования. Например, нотация диаграммы классов определяет способ представления таких элементов и понятий, как класс, ассоциация и кратность.

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

Идея строгой спецификации и языков проектирования наиболее распространена в области формальных методов. В таких методах модели и спецификации представляются с помощью некоторых производных средств исчисления предикатов. Соответствующие определения математически строги 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.

Большинство графических языков моделирования 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.

Однако методологи ищут способы добиться большей строгости методов, не жертвуя при этом их практической полезностью. Один из способов заключается в определении некоторой **метамодели** – диаграммы (как правило, диаграммы классов), определяющей понятия языка.

На рис. 1.1 изображена небольшая часть метамодели языка UML, которая показывает отношение между свойствами. (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.)

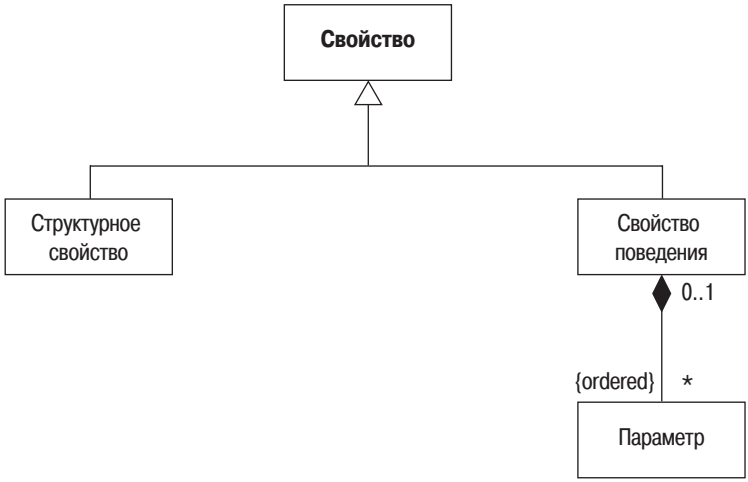

Рис. 1.1. *Фрагмент метамодели UML*

Насколько велико влияние метамодели на того, кто применяет соответствующую нотацию при моделировании? Ответ зависит, главным образом, от режима работы с языком. Создателя эскизов обычно это не слишком волнует; проектировщика это должно беспокоить значительно больше. И это жизнено важно для тех, кто использует UML в качестве языка программирования, поскольку метамодель определяет абстрактный синтаксис данного языка.

В настоящее время многие люди, вовлеченные в разработку UML, интересуются в основном метамоделью, в частности потому, что она важна при использовании UML и языка программирования. Вопросы нотации часто стоят на втором месте, что важно помнить, если вы собираетесь поближе познакомиться с документацией по стандарту.

Когда вы глубже погрузитесь в изучение UML, то поймете, что вам требуется значительно больше, чем просто графическая нотация. Вот почему инструменты UML так сложны.

В этой книге я не слишком строг. Я предпочитаю традиционные пути и обращаюсь к вашей интуиции. Это естественно для такой маленькой книжки, написанной автором, склонным, в основном, работать в режиме эскизного моделирования. Приверженцам большей строгости следует обратиться к более обстоятельным трудам.

Диаграммы UML

UML 2 описывает 13 официальных типов диаграмм, перечисленных в табл. 1.1, классификация которых приведена на рис. 1.2. Хотя эти виды диаграмм отражают различные подходы многих специалистов к UML и способ организации моей книги, авторы UML не считают диаграммы центральной составляющей языка. Поэтому диаграммы определены не очень строго. Часто вполне допустимо присутствие элементов диаграммы одного типа в другой диаграмме. Стандарт UML указывает, что определенные элементы обычно рисуются в диаграммах соответствующего типа, но это не догма.

Таблица 1.1. Официальные типы диаграмм UML

Диаграмма	Глава книги	Цель	Происхождение
Деятельности	11	Процедурное и параллельное поведение	В UML 1
Классов	3, 5	Классы, свойства и отношения	В UML 1
Взаимодействия	12	Взаимодействие между объектами; акцент на связях	Диаграмма кооперации в UML 1
Компонентов	14	Структура и взаимосвязи между компонентами	В UML 1
Составных структур	13	Декомпозиция класса во время выполнения	Новое в UML 2
Развертывания	8	Развертывание артефактов в узлы	В UML 1
Обзора взаимодействий	16	Комбинация диаграммы последовательности и диаграммы деятельности	Новое в UML 2
Объектов	6	Вариант конфигурации экземпляров	Неофициально в UML 1
Пакетов	7	Иерархическая структура времени компиляции	Неофициально в UML 1
Последовательности	4	Взаимодействие между объектами; акцент на последовательности	В UML 1
Конечных автоматов	10	Как события изменяют объект в течение его жизни	В UML 1
Временная	17	Взаимодействие между объектами; акцент на синхронизации	Новое в UML 2
Прецедентов	9	Как пользователи взаимодействуют с системой	В UML 1

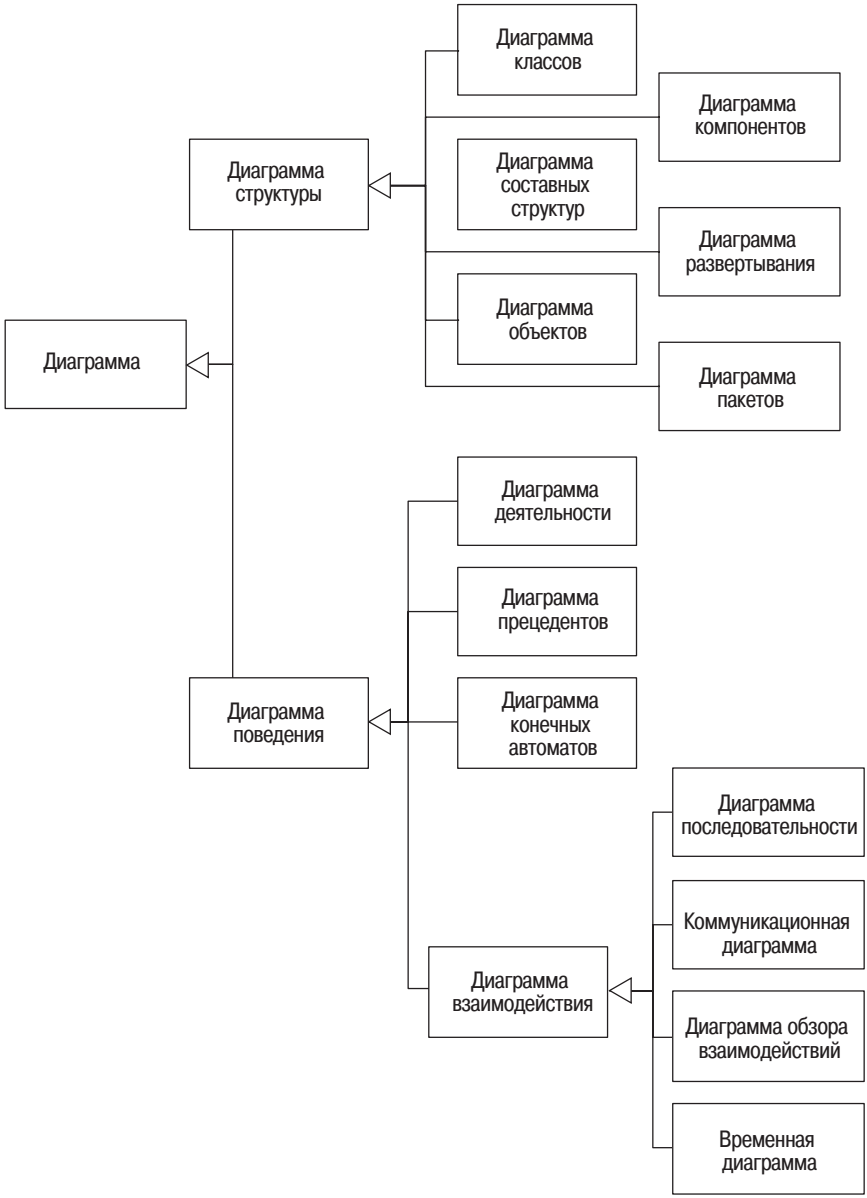

Рис. 1.2. *Классификация типов диаграмм UML*

Что такое допустимый UML?

На первый взгляд, ответить на этот вопрос легко: допустимый UML – это язык, определенный в соответствии со спецификацией. Однако на практике ответ несколько сложнее.

Существенным в вопросе является то, на каких правилах базируется UML: описательных или предписывающих. Язык с **предписывающими правилами** (prescriptive rules) управляется официальной основой, которая устанавливает, что является, а что не является допустимым языком, и какое значение вкладывается в понятие высказывания языка. Язык с **описательными правилами** (descriptive rules) – это язык, правила которого распознаются по тому, как люди применяют его на практике. Языки программирования в основном имеют предписывающие правила, установленные комитетом по стандартам или основными поставщиками, тогда как естественные языки, такие как английский, в основном имеют описательные правила, смысл которых устанавливается по соглашению.

UML – точный язык, поэтому можно было бы ожидать, что он основан на предписывающих правилах. Но UML часто рассматривают как программный эквивалент чертежей из других инженерных дисциплин, а эти чертежи основаны не на предписывающих нотациях. Никакой комитет не говорит, какие символы являются законными в строительной технической документации; эти нотации были приняты по соглашению, как и в естественном языке. Стандарты сами по себе еще ничего не решают, поскольку те, кто работает в этой области, не смогут следовать всему, что указывают стандарты; это то же самое, что спрашивать французов о французской академии наук. К тому же язык UML настолько сложен, что стандарты часто можно трактовать по-разному. Даже ведущие специалисты по UML, которые рецензировали эту книгу, не согласились бы интерпретировать стандарты.

Этот вопрос важен и для меня, пишущего эту книгу, и для вас, применяющих язык UML. Если вы хотите понять диаграммы UML, важно уяснить, что понимание стандартов – это еще не вся картина. Люди принимают соглашения и в индустрии в целом, и в каких-то конкретных проектах. Поэтому, хотя стандарт UML и может быть первичным источником информации по UML, он не должен быть единственным.

Моя позиция состоит в том, что для большинства людей UML имеет описательные правила. Стандарт UML оказывает наибольшее влияние на содержание UML, но это делает не только он. Я думаю, что особенно верным это станет для UML 2, который вводит некоторые соглашения по обозначениям, конфликтующие или с определениями UML 1, или с использованием по соглашению, а также еще больше усложняет язык. Поэтому в данной книге я стараюсь обобщить UML так, как я его вижу: и стандарты, и применение по соглашению. Когда мне приходится указывать на некоторое отличие в этой книге, я буду употреблять термин **применение по соглашению** (conventional use), чтобы обозначить то, чего нет в стандарте, но, как я думаю, широко применяется. В случае если что-то соответствует стандарту, я буду употреблять термин **стандартный** (standard) или **нормативный** (normative). (Нормативный – это термин, посредством которого люди обозначают утвер-

ждение, которое вы должны подтвердить, чтобы оно соответствовало стандарту. Поэтому выражение ненормативный UML – это своеобразный способ сказать, что нечто совершенно неприемлемо с точки зрения стандарта UML.)

Рассматривая диаграмму UML, необходимо помнить, что основной принцип UML заключается в том, что любая информация на конкретной диаграмме может быть подавлена. Это подавление может носить глобальный характер – скрыть все атрибуты – или локальный – не показывать какие-нибудь конкретные классы. Поэтому по диаграмме вы никогда не можете судить о чем-нибудь по его отсутствию. Даже если метамодель UML имеет поведение по умолчанию, например [1] для атрибутов, когда вы не видите эту информацию на диаграмме, это может быть обусловлено либо поведением по умолчанию, либо тем, что она просто подавлена.

Говоря это, следует упомянуть, что существуют основные соглашения, например о том, что многозначные свойства должны быть множествами.

Не надо слишком зацикливаться на допустимом UML, если вы занимаетесь эскизами или моделями. Важнее составить хороший проект системы, и я предпочитаю иметь хороший дизайн в недопустимом UML, чем допустимый UML, но плохой дизайн. Очевидно, хороший и допустимый предпочтительнее, но лучше направить свою энергию на разработку хорошего проекта, чем беспокоиться о секретах UML. (Конечно, в случае применения UML в качестве языка программирования необходимо соблюдать стандарты, иначе программа будет работать неправильно!)

Смысл UML

Одним из затруднений в UML является то, что хотя спецификация подробно описывает определение правильно сформированного UML, но этого недостаточно, чтобы определить значение UML вне сферы изысканного выражения «метамодель UML». Не существует формальных описаний того, как UML отображается на конкретные языки программирования. Вы не можете посмотреть на диаграмму UML и *точно* сказать, как будет выглядеть соответствующий код. Однако у вас может быть *приблизительное представление* о виде программы. На практике этого достаточно. Команды разработчиков часто формируют собственные локальные соглашения, и, чтобы их использовать, вам придется с ними познакомиться.

UML не достаточно

UML предоставляет довольно большое количество различных диаграмм, помогающих описать приложение, но это отнюдь не полный

список всех полезных диаграмм, с которыми вам, возможно, придется работать. Во многих случаях полезными могут оказаться различные диаграммы, и не надо избегать диаграмм, не имеющих отношения к UML, если не нашлось диаграмм UML, подходящих для ваших целей.

На диаграмме потока экранов (рис. 1.3) показаны различные экраны интерфейса пользователя и способы перемещения по ним. Я изучал и использовал диаграммы потока экранов многие годы и не встречал ничего, кроме очень приблизительных определений того, что они означают. В UML нет ничего подобного этим диаграммам, но я по-прежнему считаю их очень полезными.

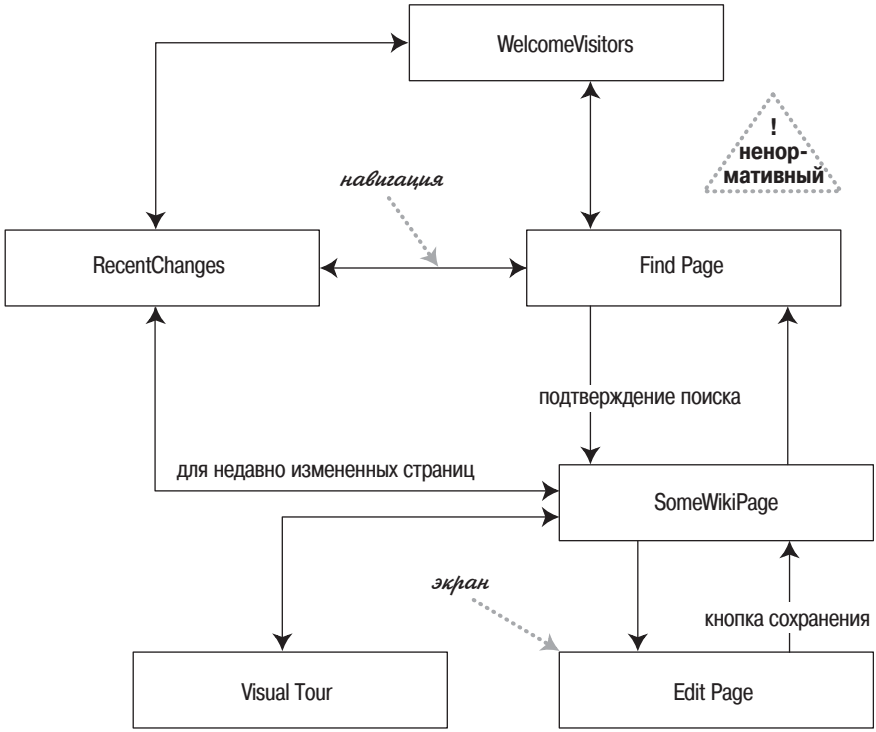

Рис. 1.3. *Неформальная диаграмма потока экранов для части wiki (http://c2.com/cgi/wiki)*

В табл. 1.2 представлен другой мой любимец – таблица решений. Таблица решений – это хороший способ показать сложные логические условия. Это можно реализовать с помощью диаграммы деятельности, но как только вы выходите за рамки простых случаев, таблица решений становится компактнее и проще для понимания. Как и диаграммы потока экранов, многие виды таблиц решений не представлены в языке. Таблица 1.2 разделена на две части: логические условия, расположенные выше двойной черты, и их результаты внизу таблицы.

Каждый столбец показывает, как конкретная комбинация условий приводит к определенному множеству результатов.

Таблица 1.2. Таблица решений

Специальный клиент	X	X	Y	Y	N	N
Приоритетный заказ	Y	N	Y	N	Y	N
Международный заказ	Y	Y	N	N	N	N
Плата	$150	$100	$70	$50	$80	$60
Предупредительный сигнал	•	•	•			

В разных книгах вы встретите различные варианты таких вещей. Не стесняйтесь пробовать приемы, которые кажутся вам подходящими для вашего проекта. Если они работают, пользуйтесь ими. Если нет – забудьте о них. (Этот же совет, конечно, относится и к диаграммам UML.)

С чего начать

Никто, даже создатели UML, не понимают всего UML и не используют все его возможности. Большинство разработчиков при работе задействуют лишь небольшое подмножество UML. Вы должны найти свое подмножество, которое бы подходило для решения задач, стоящих перед вами и вашими коллегами.

Тем, кто только начинает, я советую на первых порах сконцентрироваться на основных формах диаграмм классов и диаграмм последовательности. На мой взгляд, это наиболее общие и самые полезные типы диаграмм.

Постигнув их суть, вы сможете приступить к применению более продвинутых нотаций диаграмм классов и взглянуть на другие типы диаграмм. Экспериментируйте с диаграммами и выясните, насколько они полезны для вас. Не бойтесь отбросить любые из них, которые не кажутся вам полезными для вашей работы.

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. По ходу рассмотрения отдельных тем будут упоминаться и другие книги, к которым следует обратиться для получения более подробной информации. Будут упомянуты некоторые основные книги о языке UML и объектно-ориентированном проектировании.

Рассматривая рекомендации всех книг, необходимо проверять версии UML, для которых они написаны. Что касается июня 2003 года, то на этот момент нет опубликованных книг, использующих UML 2.0, что вряд ли может удивить, поскольку едва-едва высохли чернила на тексте стандарта. Предлагаемые мной книги хороши, но я не могу утверждать, что они будут обновлены в соответствии со стандартом UML 2.

Если вы новичок в обращении с объектами, то я рекомендую обратиться к моей любимой книге – [29]. Стоит принять на вооружение подход автора к проектированию, который основан на строгом управлении ответственностями.

И в качестве заключительного источника информации по UML посмотрите официальные документы по стандартам, но помните, что они написаны в соответствии с представлениями методологов в уединении их личных каморок. Более удобоваримую версию стандарта можно посмотреть в книге Рамбо [40].

Если вам нужна более подробная информация по объектно-ориентированному проектированию, то много интересного вы найдете в книге Мартина [30].

Я также предлагаю почитать книги по шаблонам, что позволит вам выйти за пределы основ. Теперь, когда война методов закончена, именно в области шаблонов (*стр. 54*) появляются интересные материалы по анализу и дизайну.

2

Процесс разработки

Как я уже говорил, UML вырос из группы методов объектно-ориентированного анализа и дизайна. До некоторой степени все они представляют собой комбинацию графического языка моделирования и процесса, в котором определяются подходы к разработке программного обеспечения.

Интересно, что когда UML сформировался, многие участники обнаружили, что хотя они и могут принять язык моделирования, но определенно не могут согласиться с процессом. В результате они согласились перенести все соглашения по процессу на более позднее время и ограничить применение UML областью языка моделирования.

Эта книга называется «UML. Основы», поэтому я могу оставить в покое проигнорированный процесс. Однако я не думаю, что приемы моделирования имеют смысл без понимания того, как они соответствуют процессу. Способы применения UML в значительной степени зависят от типа процесса, с которым вы работаете.

Таким образом, я считаю, что необходимо сначала поговорить о процессе, чтобы вы смогли увидеть контекст использования UML. Я не собираюсь подробно описывать какой-либо конкретный процесс; я просто хочу дать вам достаточно информации, чтобы вы смогли увидеть этот контекст, и показать, где вы можете узнать больше.

В беседах о UML можно часто услышать упоминание о RUP (Rational Unified Process – унифицированный процесс, созданный компанией Rational). RUP – это один из процессов, точнее говоря, процесс-структура, который вы можете использовать с UML. Но кроме общего участия различных сотрудников фирмы Rational и названия «унифицированный», он не имеет особенного отношения к UML. Язык UML можно использовать с любым процессом. RUP является популярным подходом; он обсуждается на *стр. 52*.

Процессы итеративные и водопадные

Самая бурная дискуссия о процессах разворачивается вокруг выбора между итеративной и водопадной моделями. Часто эти термины употребляются неправильно, в частности потому, что применение итеративного процесса вошло в моду, а водопадный процесс считается чем-то вроде брюк в клеточку. В результате процесс разработки многих проектов объявляется итеративным, хотя в действительности он является процессом водопадного типа.

Существенная разница между двумя этими типами проявляется в том, каким образом проект делится на более мелкие части. Если предполагается, что разработка проекта займет год, то мало кто сможет с легким сердцем сказать людям, что им надо уйти на год и вернуться, когда все будет сделано. Необходимо некоторое разделение, чтобы разработчики смогли найти подход к решению проблемы и наладить работу.

При организации работы в стиле **водопада** проект делится на основании вида работ. Чтобы создать программное обеспечение, необходимо предпринять определенные действия: проанализировать требования, создать проект, выполнить кодирование и тестирование. Наш годичный проект может включать двухмесячную фазу анализа, за которой следует четырехмесячная фаза дизайна, а затем трехмесячная фаза кодирования и, наконец, трехмесячная фаза тестирования.

Итеративный стиль делит проект по принципу функциональности продукта. Можно взять год и разделить его на трехмесячные итерации. В первой итерации берется четверть требований и выполняется полный цикл разработки программного обеспечения для этой четверти: анализ, дизайн, кодирование и тестирование. К концу первой итерации у вас есть система, обладающая четвертью необходимой функциональности. Затем вы приступаете ко второй итерации и через шесть месяцев получаете систему, делающую половину того, что ей положено.

Естественно, приведенное выше описание упрощено, но в этом состоит суть различия. Конечно, на практике к течению процесса добавляются непредвиденные вредные ручейки.

При разработке способом водопада после каждого этапа обычно в каком-либо виде выполняется формальная сдача, но часто имеет место возвращение назад. В процессе кодирования могут выясниться обстоятельства, вынуждающие снова вернуться к этапам анализа и дизайна. Конечно, в начале кодирования не следует думать, что анализ завершен. И решения, принятые на стадии анализа и дизайна, неизбежно будут пересматриваться позднее. Однако эти обратные потоки представляют собой исключения и должны быть по возможности сведены к минимуму.

При итеративном процессе разработки перед началом реальной итерации обычно наблюдается некоторая исследовательская активность.

Как минимум на требования будет брошен поверхностный взгляд, достаточный, по крайней мере, для разделения требований на итерации для последующего выполнения. В процессе такого исследования могут быть приняты некоторые решения по дизайну самого высшего уровня. С другой стороны, несмотря на то что в результате каждой итерации должно появиться интегрированное программное обеспечение, готовое к поставке, часто бывает, что оно еще не готово и нужен некоторый стабилизационный период для исправления последних ошибок. Кроме того, некоторые виды работ, такие как тренировка пользователей, оставляются на конец.

Конечно, вы вполне можете не передавать систему на реализацию в конце каждой итерации, но она должна находиться в состоянии производственной готовности. Однако часто бывает, что система сдается на регулярной основе; это хорошо, поскольку вы оцениваете работоспособность системы и получаете более качественную обратную реакцию. В этой ситуации часто говорят о проекте, имеющем несколько версий, каждая из которых делится на несколько **итераций**.

Итеративную разработку называют по-разному: инкрементной, спиральной, эволюционной и постепенной. Разные люди вкладывают в эти термины разный смысл, но эти различия не имеют широкого признания и не так важны, как противостояние итеративного метода и метода водопада.

Возможен и смешанный подход. В книге Мак-Коннелла [31] описывается жизненный цикл **поэтапной доставки** (staged delivery), в соответствии с которым сначала выполняются анализ и проектирование верхнего уровня в стиле водопада, а затем кодирование и тестирование, разделенные на итерации. В таком проекте может быть четырехмесячный этап анализа и дизайна, а затем четыре двухмесячные итерации построения системы.

Большинство авторов публикаций по процессу создания программного обеспечения, особенно принадлежащие к объектно-ориентированному сообществу, последние пять лет испытывают неприязнь к подходу в стиле водопада. Из всего множества причин этого явления самая главная заключается в том, что при использовании метода водопада очень трудно утверждать, что разработка какого-то проекта действительно идет в верном направлении. Слишком легко объявить победу на раннем этапе и скрыть ошибки планирования. Обычно единственный способ, которым вы действительно можете показать, что следуете по такому пути, состоит в том, чтобы получить протестированное, интегрированное программное обеспечение. В случае итеративного процесса это повторяется многократно, и в результате, если что-то идет не так, как надо, мы своевременно получаем соответствующий сигнал.

Хотя бы только по этой причине я настоятельно рекомендую избегать метода водопада в чистом виде. По крайней мере, необходимо приме-

нять поэтапную доставку, если невозможно использовать итеративный метод в полном объеме.

Объектно-ориентированное сообщество долгое время было приверженно итеративному способу разработки, и, не боясь ошибиться, можно сказать, что значительная часть разработчиков, участвующих в создании UML, предпочитают в том или ином виде итеративную разработку. Однако, по моим ощущениям, в практике программной индустрии метод водопада все еще занимает ведущее положение. Одну из причин этого явления я называю псевдоитеративной разработкой: исполнители объявляют, что ведут итеративное проектирование, а на самом деле действуют в стиле водопада. Основные симптомы этого следующие:

- «Мы выполняем одну итерацию анализа, а затем две итерации проектирования...»
- «Код данной итерации содержит много ошибок, но мы исправим их в конце...»

Особенно важно, чтобы каждая итерация завершалась созданием протестированного, интегрированного программного продукта, который бы имел качество, как можно более близкое к качеству серийной продукции. Тестирование и интеграцию оценить труднее всего, поэтому в конце разработки проекта лучше эти этапы не проводить. Процесс тестирования следует организовать так, чтобы любая итерация, не объявленная в плане как сдаточная, могла бы быть переведена в такой статус без серьезных дополнительных усилий разработчиков.

Общим приемом при итеративной разработке является **упаковка по времени** (time boxing). Таким образом, итерация будет занимать фиксированный промежуток времени. Если обнаружилось, что вы не в состоянии выполнить все, что планировали сделать за время итерации, то необходимо выбросить некоторую функциональность из данной итерации, но не следует изменять дату исполнения. В большинстве проектов, основанных на итеративном процессе, протяженность итераций одинакова, и это позволяет вести разработку в постоянном ритме.

Мне нравится упаковка по времени, поскольку люди обычно испытывают трудности при сокращении функциональности. Регулярно практикуясь в сокращении функциональности, они учатся делать осмысленный выбор между изменением времени разработки и изменением функциональности. Сокращение функциональности в ходе итерации позволяет также людям научиться расставлять приоритеты между требованиями к проекту.

Одним из наиболее общих аспектов итеративной разработки является вопрос переделки. Итеративная разработка недвусмысленно предполагает, что вы будете перерабатывать и удалять существующий код на последней итерации проекта. Во многих областях человеческой деятельности, например в промышленном производстве, переделка считается ущербом. Но создание программного обеспечения не похоже на

промышленное производство – часто бывает выгоднее переработать существующий код, чем латать код неудачно спроектированной программы. Некоторые технические приемы способны оказать существенную помощь, чтобы процесс переделки был более эффективным.

- **Автоматизированные регрессивные тесты** (automated regression tests) позволяют быстро найти любые ошибки, внесенные в процессе изменений. Семейство оболочек тестирования xUnit представляет наиболее подходящий инструмент для создания автоматизированных тестов для модульной проверки (unit tests). Начиная с исходного JUnit *http://junit.org*, здесь есть порты для почти всех возможных языков (*http://www.xprogramming.com/software.htm*). Хорошим правилом является создание модульных тестов примерно такого же размера, что и тестируемая программа.
- **Рефакторинг** (refactoring) – это способ изменения существующего программного продукта [20]. Механизм рефакторинга основан на применении серии небольших, сохраняющих поведение трансформаций основного кода. Многие из этих трансформаций могут быть автоматизированы (*http://www.refactoring.com*).
- **Последовательная интеграция** (continuous integration) сохраняет синхронизацию действий разработчиков, объединенных в команду, что позволяет избежать болезненных циклов интеграции [18]. В ее основе лежит полностью автоматизированный процесс построения, который может быть автоматически прекращен, как только любой член команды начнет записывать код в базу. Предполагается, что разработчики записывают код ежедневно, поэтому автоматические сборки выполняются несколько раз в день. Процесс построения включает запуск большого блока автоматических регрессионных тестов, что позволяет быстро обнаружить и без труда исправить любую несогласованность.

Все эти технические приемы пропагандировались недавно в книге «Extreme Programming» [2], хотя они применялись и раньше и могли (и должны были) применяться, независимо от того, использовался ли XP (eXtreme Programming) или какой-либо другой гибкий процесс.

Прогнозирующее и адаптивное планирование

Одна из причин, по которым метод водопада еще жив, заключается в желании обеспечить предсказуемость при создании программного обеспечения. Ничто так не раздражает, как отсутствие точной оценки стоимости создания программного продукта и сроков его разработки.

Прогнозирующий подход направлен на выполнение работы на начальном этапе проекта, для того чтобы лучше понять, что нужно делать в дальнейшем. Таким образом, наступает момент, когда оставшуюся часть проекта можно оценить с достаточной степенью точности. В процессе **прогнозирующего планирования** (predictive planning) проект

разделяется на две стадии. На первой стадии составляются планы, и тут предсказывать трудно, но вторая стадия более предсказуема, поскольку планы уже готовы.

При этом не надо все делить на белое и черное. В процессе выполнения проекта вы постепенно добиваетесь большей предсказуемости. И даже если у вас есть план, все может пойти не так, как вы спрогнозировали. Вы просто ожидаете, что при наличии четкого плана отклонения будут менее значительными.

Однако все еще идут острые дискуссии о том, много ли проектов могут быть предсказуемыми. Сущность данного вопроса состоит в анализе требований. Одна из самых существенных причин сложности программных проектов заключается в трудности понимания требований к программным системам. Большинство программных проектов подвергаются существенному **пересмотру требований** (requirements churn): изменению требований на поздней стадии выполнения проекта. Такой пересмотр вдребезги разбивает основу прогнозов. Последствия пересмотра можно предотвратить, заморозив требования на ранней стадии проекта и не позволяя изменениям появляться, но это приводит к риску поставить клиенту систему, которая больше не удовлетворяет требованиям пользователей.

Эта проблема приводит к двум различным вариантам действий. Один путь – это направить больше усилий собственно на проработку требований. Другой путь состоит в получении более определенного множества требований, чтобы сократить возможные изменения.

Приверженцы другой школы утверждают, что пересмотр требований неизбежен, что во многих проектах трудно стабилизировать требования в такой степени, чтобы имелась возможность использовать прогнозирующее планирование. Это может быть либо следствием того, что исключительно трудно представить, что может делать программный продукт, либо следствием того, что условия рынка диктуют непредсказуемые изменения. Эта школа поддерживает **адаптивное планирование** (adaptive planning) в соответствии с утверждением, что прогнозируемость – это иллюзия. Вместо того чтобы дурачить себя иллюзорной предсказуемостью, мы должны повернуться лицом к реальности постоянных изменений и использовать такой подход в планировании, при котором изменение в проекте считается величиной постоянной. Это изменение контролируется таким образом, чтобы в результате выполнения проекта поставлялось как можно лучшее программное обеспечение; но хотя проект и контролируем, предсказать его нельзя.

Различие между прогнозирующими и адаптивными проектами проявляется разными путями, когда люди говорят о состоянии проекта. Когда утверждается, что выполнение проекта идет хорошо, поскольку работа ведется в соответствии с планом, то имеется в виду метод прогнозирования. При адаптивной разработке нельзя сказать «в соответствии с планом», поскольку план все время меняется. Это не означает, что

адаптивные проекты не планируются; обычно планирование занимает значительное время, но план трактуется как основная линия проведения последовательных изменений, а не как предсказание будущего.

На основе прогнозирующего плана можно разработать контракт с фиксированной функциональностью по фиксированной цене. В таком контракте точно указывается, что должно быть создано, сколько это стоит и когда продукт будет поставлен. В адаптивном плане такое фиксирование невозможно. Вы можете обозначить бюджет и сроки поставки, но вы не можете точно зафиксировать функциональность поставляемого продукта. Адаптивный контракт предполагает, что пользователи будут сотрудничать с командой разработчиков, чтобы регулярно пересматривать требуемую функциональность и прерывать проект, если прогресс слишком незначителен. Как таковой процесс адаптивного планирования может определять проект с переменными границами функциональности по фиксированной цене.

Естественно, адаптивный подход менее желателен, поскольку все предпочитают большую предсказуемость программных проектов. Однако предсказуемость зависит от точности, корректности и стабильности множества требований. Если вы не в состоянии стабилизировать свои требования, то прогнозирующий план базируется на песке, а изменения настолько значительны, что проект сбивается с курса. Отсюда вытекают два важных совета.

1. Не составляйте прогнозирующий план до тех пор, пока не получите точные и корректные требования и не будете уверены, что они не подвергнутся существенным изменениям.
2. Если вы не можете получить точные, корректные и стабильные требования, то используйте метод адаптивного планирования.

Предсказуемость и адаптивность предоставляют выбор жизненного цикла. Адаптивное планирование совершенно определенно подразумевает итеративный процесс. Прогнозирующий план может быть выполнен любым из двух способов, хотя за его выполнением легче наблюдать в случае применения метода водопада, или метода поэтапной поставки.

Гибкие процессы

За последние несколько лет вырос интерес к гибким процессам разработки программного обеспечения. *Гибкий* (*agile*) – это широкий термин, охватывающий большое количество процессов, имеющих общее множество величин и понятий, определенных Манифестом гибкой разработки программного обеспечения (Manifesto of Agile Software Development) (*http://agileManifesto.org*). Примерами таких процессов являются XP (Extreme Programming – экстремальное программирование), Scrum (столкновение), FDD (Feature Driven Development – разработка, управляемая возможностями), Crystal (кристалл) и DSDM (Dynamic Systems Development Method – метод разработки динамических систем).

В терминах нашего обсуждения гибкие процессы исключительно адаптивны по своей природе. Они также имеют четкую ориентацию на человека. Гибкие подходы предполагают, что наиболее важным фактором успешного завершения проекта является квалификация исполнителей и их хорошая совместная работа с человеческой точки зрения. Значимость процессов или инструментов, ими используемых, определенно стоит на втором месте.

Гибкие методы в основном направлены на использование коротких, ограниченных по времени итераций, чаще всего заканчивающихся через месяц или раньше. Поскольку их вклад в документацию невелик, то в гибком подходе не предполагается применение UML в режиме проектирования. Чаще всего UML используется в режиме эскизирования и реже в качестве языка программирования.

В большинстве своем гибкие процессы не слишком **формализованы**. Сильно формализованные или тяжеловесные процессы имеют много документации и постоянный контроль во время выполнения проекта. Гибкий подход предполагает, что формализм мешает проведению изменений и противоречит природе талантливых личностей. Поэтому гибкие процессы часто называют **облегченными** (lightweight). Важно понимать, что недостаточная формализованность является следствием адаптивности и ориентации специалистов, а не фундаментальным свойством.

Унифицированный процесс от Rational

Хотя унифицированный процесс, разработанный компанией Rational (Rational Unified Process, RUP), не зависит от UML, их часто упоминают вместе. Поэтому я думаю, что будет уместно сказать здесь об этом несколько слов.

Хотя RUP называется процессом, в действительности это оболочка процессов, предоставляющая словарь и свободную структуру для обсуждения процессов. В случае применения RUP в первую очередь необходимо выбрать **шаблон разработки** (development case) – процесс, который вы собираетесь использовать в проекте. Шаблоны разработки могут очень значительно варьироваться, поэтому не думайте, что ваш шаблон разработки будет сильно похож на другие шаблоны. При выборе шаблона разработки сразу требуется человек, хорошо знакомый с RUP, – тот, кто сможет приспособить RUP к определенным требованиям проекта. В качестве альтернативы существует постоянно увеличивающийся набор распределенных по пакетам шаблонов разработки, с которых можно начать.

Независимо от шаблона разработки RUP по существу является итеративным процессом. Метод водопада не совместим с философией RUP, хотя с прискорбием должен отметить, что проекты, в которых применяются процессы в стиле водопада, нередко обряжают в одежды RUP.

Все RUP-проекты должны иметь четыре фазы.

1. **Начало** (inception). На этой стадии осуществляется первичная оценка проекта. Обычно именно здесь вы решаете, стоит ли вкладывать средства в фазу уточнения.
2. **Уточнение (**elaboration**).** На этой стадии идентифицируются основные прецеденты проекта и в итеративном процессе создается программное обеспечение, для того чтобы развернуть архитектуру системы. В конце фазы уточнения у вас должно быть достаточно полное понимание требований и скелет работающей системы, которую можно взять за основу разработки. В частности, необходимо обнаружить и разрешить основные риски проекта.
3. На стадии **построения (**construction**)** продолжается процесс создания и разрабатывается функциональность, достаточная для выпуска продукта.
4. **Внедрение** (transition) состоит из различных стадий работы, выполняемых в конце и в неитеративном режиме. Они могут включать развертывание в информационном центре, обучение пользователей и тому подобное.

Между фазами существует полная неопределенность, особенно между уточнением и построением. Для кого-то переход к построению – это момент, когда можно переключиться в режим прогнозирующего планирования. А для кого-то это просто точка, в которой появляется ясное понимание требований и архитектуры, определение которой, как вам кажется, движется к завершению.

Иногда RUP называют просто унифицированным процессом (Unified Process, UP). Так обычно поступают организации, которые хотят применить терминологию и общий подход RUP, но не хотят пользоваться лицензионными продуктами фирмы Rational Software. Можно думать о RUP как о продукте фирмы Rational, основанном на UP, а можно считать RUP и UP одним и тем же. В обоих случаях вы найдете людей, которые с вами согласятся.

Настройка процесса под проект

Программные проекты значительно отличаются друг от друга. Способ, которым ведется разработка программного обеспечения, зависит от многих факторов: типа создаваемой системы, используемой технологии, размера и распределенности команды, характера рисков, последствий неудач, стиля работы в команде и культуры организации. Поэтому не следует ожидать, что найдется процесс, подходящий для всех проектов.

Следовательно, необходимо приспособить процесс, чтобы он соответствовал вашему конкретному окружению. Один из первых шагов, которые необходимо сделать, – это взглянуть на проект и выбрать наиболее

подходящие процессы. Таким образом, вы получите короткий список процессов.

Затем необходимо определить, что следует предпринять, чтобы настроить процесс под конкретный проект. При этом надо соблюдать осторожность. Некоторые процессы трудно оценить, не поработав с ними. В таких случаях лучше провести с новым процессом пару итераций, чтобы понять, как он функционирует. Затем можно начинать модифицировать процесс. Если вы с самого начала знаете, как работает процесс, то можно модифицировать его без предварительной подготовки. Помните, что, как правило, легче начинать с малого и понемногу добавлять, чем сделать слишком много, а потом что-то выбрасывать.

Шаблоны

UML говорит, как изобразить объектно-ориентированный дизайн. Напротив, шаблоны представляют результат: примеры дизайна.

Многие утверждают, что в процессе выполнения проектов появляются трудности, поскольку исполнители не так сведущи в дизайне, который хорошо известен более подготовленным разработчикам. Шаблоны описывают общие подходы к работе, и они собираются людьми, которые обнаруживают повторяющиеся темы в процессе дизайна. Эти люди берут каждую такую тему и описывают ее так, чтобы другие разработчики смогли прочитать шаблон и узнать способ его применения.

Рассмотрим пример. Предположим, что у вас есть некоторые объекты, запускаемые в процессе на рабочем столе, и они должны взаимодействовать с другими объектами, запускаемыми во втором процессе. Возможно, второй процесс располагается также на вашем рабочем столе; возможно, он располагается в другом месте. Объекты вашей системы не должны искать другие процессы в сети или выполнять удаленные вызовы процедур.

Для удаленного объекта можно создать объект-посредник внутри локального процесса. Посредник имеет тот же самый интерфейс, что и удаленный объект. Локальный объект общается с посредником при помощи рассылки обычных сообщений внутри процесса. При этом посредник отвечает за передачу сообщений реальному объекту независимо от его местоположения.

Создание посредника – это обычная практика в сетевом взаимодействии и в других областях. Специалисты имеют большой опыт работы с посредниками, знают, как их применять, какие это дает преимущества и как их реализовать, какие им свойственны ограничения. Эти навыки обсуждаются в методических изданиях, подобных этому; все они рассказывают, как можно схематически представить посредника, и хотя это полезно, но не так, как было бы полезно рассмотрение опыта использования посредника.

В начале 90-х годов специалисты начали приобретать такой опыт. Они образовали сообщество людей, заинтересованных в написании шаблонов. Эти специалисты спонсировали ряд конференций и выпустили несколько книг.

Наиболее известной публикацией по шаблонам, выпущенной этой группой, является книга «банды четырех» [21], в которой подробно рассмотрены 23 шаблона дизайна. Посредникам в этой книге посвящен десяток страниц, дающих детальное представление о том, как объекты работают друг с другом; кроме того, обсуждаются преимущества и ограничения шаблонов, общие варианты использования и советы по реализации шаблонов.

Шаблон – это больше, чем модель. Шаблон должен также включать обоснование выбранного пути. Часто говорят, что шаблон – это ключ к решению проблемы. Шаблон должен четко идентифицировать проблему, объяснить, почему он решает проблему, а также объяснить, при каких условиях шаблон работает, а при каких нет.

Шаблоны важны, поскольку они являются следующим этапом в понимании основ языка или технологии моделирования. Шаблоны предоставляют набор решений, а также показывают, что помогает создать хорошую модель и какова последовательность действий при разработке модели. Шаблоны учат на примерах.

Когда я начинал, меня удивляло, почему я должен все изобретать с нуля. Почему у меня нет руководства, в котором бы рассказывалось о том, как делать общие вещи? Сообщество пытается создать такую книгу.

В настоящее время издано множество книг по шаблонам, и они очень отличаются по качеству. Мои любимые – это [21], [36], [37], [13], [35] и, извините за нескромность, [16] и [19]. Кроме того, вы можете зайти на домашнюю страничку шаблонов: *http://www.hillside.net/patterns*.

Как бы ни были вы вначале уверены, что знаете свой процесс, очень важно, чтобы по мере продвижения вперед вы учились. Действительно, одно из больших преимуществ итеративной разработки состоит в возможности часто совершенствовать процесс.

В конце каждой итерации следует проводить ее **ретроспективный анализ** (iteration retrospective), собирая команду на совещания, чтобы рассмотреть, как идут дела и что можно улучшить. Если итерация короткая, то достаточно двух часов. При этом хорошо составить список из трех категорий:

1. *Сохранить*: все, что работало правильно, и вы хотите это продолжить.
2. *Проблемы*: разделы, которые работали неправильно.
3. *Испытать*: изменения в процессе с целью его улучшения.

Вы можете начинать ретроспективный анализ каждой итерации, рассматривая элементы предыдущей сессии и определяя их изменения. Не забудьте про список того, что нужно сохранить; важно отслеживать элементы, которые работают правильно. Если вы этого не делаете, то можете потерять ощущение перспективы проекта и, возможно, перестать обращать внимание на успешные приемы.

В конце разработки проекта или его основной версии можно провести более формальный **ретроспективный анализ проекта** (project retrospective), который может занять пару дней; более подробную информацию можно найти на *http://www.retrospectives.com/* и в книге [26]. Больше всего меня раздражает, когда организации упорно игнорируют собственный опыт и постоянно совершают одни и те же ошибки.

Настройка UML под процесс

При рассмотрении графических языков моделирования обычно о них думают в контексте водопадного процесса. Водопадный процесс, как правило, сопровождается документами, выступающими в качестве пресс-релизов между стадиями анализа, дизайна и кодирования. Часто графические модели могут занимать основную часть этих документов. Действительно, многие из структурных методов 70-х и 80-х годов часто говорят о моделях анализа и дизайна, подобных этой.

Независимо от того, применяете вы метод водопада или нет, так или иначе вы проводите анализ, дизайн, кодирование и тестирование. Можно запустить итеративный проект с недельными итерациями, когда каждую неделю работает метод водопада.

Использование UML не подразумевает обязательную разработку документов или загрузку сложных CASE-систем.

Анализ требований

В процессе анализа требований необходимо понять, что клиенты и пользователи программного обеспечения ожидают от системы. В вашем распоряжении имеется множество приемов UML:

- Прецеденты, которые описывают, как люди взаимодействуют с системой.
- Диаграмма классов, которая строится с точки зрения концептуальной перспективы и может служить хорошим инструментом для построения точного словаря предметной области.
- Диаграмма деятельности, которая показывает рабочий поток организации, способы взаимодействия программного обеспечения и пользователей. Диаграмма деятельности может показать контекст для использования прецедентов, а также детали работы сложных прецедентов.

- Диаграмма состояний, которая может оказаться полезной, если концепция имеет своеобразный жизненный цикл с различными состояниями и событиями, которые изменяют эти состояния.

Анализируя состояния, помните, что самое важное – это взаимодействие с вашими пользователями и клиентами. Обычно это непрограммисты, и они не знакомы с UML и другими подобными технологиями. Несмотря на это я успешно применял перечисленные приемы при общении с людьми, не имеющими инженерной подготовки. Чтобы добиться этого, надо свести количество нотаций к минимуму. Не следует вводить элементы, специфичные для программной реализации.

Будьте готовы в любой момент отойти от правил UML, если это поможет улучшить взаимопонимание. Наибольший риск в случае применения UML для анализа состоит в том, что вы строите диаграммы, не совсем понятные специалистам в конкретной предметной области. Такая диаграмма хуже, чем бесполезная; она лишь способна вселить в разработчиков ложное чувство уверенности.

Проектирование

При разработке модели вы можете широко применять диаграммы. Можно использовать больше нотаций и при этом быть более точным. Вот некоторые полезные приемы:

- Диаграммы классов с точки зрения программного обеспечения. Они показывают классы программы и их взаимосвязи.
- Диаграммы последовательности для общих сценариев. Правильный подход состоит в извлечении наиболее важных и интересных сценариев из прецедента, а также в использовании CRC-карточек и диаграмм последовательности с целью понять, что происходит в программе.
- Диаграммы пакетов, показывающие высокоуровневую организацию программного продукта.
- Диаграммы состояний для классов со сложным жизненным циклом.
- Диаграммы развертывания, показывающие физическую конфигурацию программного обеспечения.

Многие из этих приемов позволяют документировать программное обеспечение после его создания. Кроме того, это может помочь людям сориентироваться в программе, если они ее не создавали и не знакомы с кодом.

В рамках жизненного цикла метода водопада необходимо создавать эти диаграммы и выполнять различного рода действия (активности) как часть конкретных фаз. Документы, создаваемые по окончании какой-либо фазы, обычно включают диаграммы для проведенных действий. Стиль водопада подразумевает применение UML в качестве инструмента проектирования.

При итеративном подходе диаграммы UML могут выступать или как модели, или как эскизы. В режиме проектирования аналитические диаграммы обычно создаются в рамках итерации, предшествующей итерации построения функциональности. Каждая итерация не начинается с самого начала. Напротив, она модифицирует существующую документацию, подчеркивая изменения, произошедшие в новой итерации.

Создание моделей обычно происходит на ранней стадии итерации и может быть сделано по частям для различных разделов функциональности, назначенной для данной итерации. Кроме того, итерация подразумевает изменение существующей модели, а не построение каждый раз новой модели.

Применение UML в режиме эскизирования – более подвижный процесс. Один из подходов заключается в выделении пары дней в начале итерации на создание эскизного дизайна для данной итерации. Можно также проводить короткие сессии проектирования в любой момент в ходе итерации, устраивая короткие получасовые совещания всякий раз, когда разработчики начинают спорить по поводу нетривиальной функции.

В режиме проектирования вы ожидаете, что программная реализация будет строиться в соответствии с диаграммами. Изменение модели следует считать отклонением, которое требует, чтобы проектировщики, создавшие эту модель, ее пересмотрели. Эскиз обычно трактуется как первый срез дизайна. Если в ходе кодирования обнаруживается, что эскиз не совсем точен, то разработчики должны иметь возможность изменить дизайн. Разработчики, внедряющие дизайн, должны сами решать, стоит ли устраивать широкую дискуссию, чтобы понять все возможные варианты.

Относительно моделей у меня есть свое собственное мнение, которое заключается в том, что даже хорошему проектировщику очень трудно составить правильную модель. Я часто обнаруживаю, что моя собственная модель не остается нетронутой по завершении кодирования. И все же я считаю эскизы UML полезными, хотя не думаю, что их следует возводить в абсолют.

В обоих режимах имеет смысл исследовать несколько вариантов дизайна. Как правило, лучше просматривать варианты в режиме эскизного моделирования, чтобы иметь возможность быстро создавать и изменять варианты. Выбрав дизайн, вы можете либо использовать эскиз, либо детально проработать его до уровня модели.

Документация

После создания программного обеспечения можно написать документацию на готовый продукт с помощью UML. Я считаю, что диаграммы UML очень помогают понять систему в целом. Однако я должен под-

черкнуть, что не верю в возможность создания подробных диаграмм всей системы. Процитирую Уорда Каннингема [14]:

> *Тщательно отобранные и хорошо написанные заметки могут легко заменить традиционную обширную документацию. Последняя редко проясняет суть вещей, за исключением отдельных аспектов. Выделите эти аспекты … и забудьте обо всем остальном.*

Я полагаю, что подробная документация должна генерироваться на основе программного кода – так, как это делает, например JavaDoc. Для того чтобы подчеркнуть важные концепции, необходимо написать дополнительную документацию. Можно считать ее составной частью первого этапа знакомства с программой, предшествующего детальному изучению кода. Я предпочитаю представлять документацию в виде текста, достаточно краткого, чтобы его можно было прочитать за чашкой кофе, и снабженного диаграммами UML, придающими обсуждению наглядный характер. Очевидно, что составитель документации должен решать, что важно, а что нет, поскольку разработчик вооружен для этого значительно лучше, чем читатель.

Диаграмма пакетов представляет хорошую логическую маршрутную карту системы. Эта диаграмма помогает понять логические блоки системы, а также обнаружить их взаимозависимости и держать их под контролем. Диаграмма развертывания (см. главу 8), которая показывает физическую картину системы на верхнем уровне, также может оказаться полезной на этой стадии.

Для каждого пакета я предпочитаю строить диаграмму классов. При этом я не указываю каждую операцию в том или ином классе, а показываю только важные свойства, которые помогают мне понять общую картину. Такая диаграмма классов служит своего рода оглавлением в виде графической таблицы.

Диаграмму классов следует сопроводить несколькими диаграммами взаимодействий системы, которые показывают наиболее важные из них. Повторюсь: правильный отбор очень важен; помните, что в документации такого типа полнота – враг понятности.

Если некоторый класс в течение своего жизненного цикла имеет сложное поведение, то для его описания я строю диаграмму конечного автомата (глава 10). Но делаю это только в случае достаточно сложного поведения, что, на мой взгляд, бывает очень редко.

В книге также часто встречаются фрагменты программного кода, важные для понимания системы и профессионально написанные. Кроме того, я использую диаграмму деятельности (глава 11), но только если она обеспечивает мне лучшее понимание, чем сам код.

Сталкиваясь с повторяющимися понятиями, я описываю их при помощи шаблонов (*стр. 54*).

В документации важно отразить неиспользованные варианты дизайна и пояснить, почему они были отброшены. Об этом часто забывают, но

такая информация может оказаться самой важной для ваших пользователей.

Понимание унаследованного кода

Язык UML помогает вникнуть за пару дней в труднодоступную для понимания ветвь незнакомой программы. Построение эскиза ключевых аспектов системы может действовать как графический запоминающий механизм, который помогает зафиксировать важную информацию о системе в процессе ее изучения. Эскизы ключевых классов и их наиболее важных взаимосвязей помогают пролить свет на происходящее.

Современные инструменты позволяют генерировать подробные диаграммы ключевых разделов системы. Не следует применять эти инструменты для создания больших бумажных отчетов; лучше с их помощью вскрывать наиболее важные пласты исследуемого программного кода. Особенно радует, что есть возможность генерации диаграмм последовательности, позволяющих проследить взаимодействие множества объектов при реализации сложного метода.

Выбор процесса разработки

Я твердый сторонник итеративного процесса разработки. Как я уже говорил в этой книге: «Применяйте итеративный метод разработки только в проектах, которым вы желаете успеха».

Может быть, кому-то покажется, что это болтовня, но с годами я становлюсь все более агрессивным сторонником итеративной разработки. При грамотном применении она является весьма важным методом, способным помочь в раннем выявлении возможных рисков и в улучшении управляемости процессом разработки. Однако это не означает, что можно вовсе обойтись без руководства проектом (хотя, если быть справедливым, я должен отметить, что некоторые используют ее именно для этой цели). Итеративная разработка требует тщательного планирования. Но это весьма надежный подход, и поэтому любая книга по объектно-ориентированной разработке рекомендует его применять – и не без основания.

Вы не должны удивляться, услышав, что я – как один из авторов Манифеста по гибкой разработке программного обеспечения (Manifesto for Agile Software Development) – большой любитель гибких подходов. У меня также накоплен большой положительный опыт в экстремальном программировании (Extreme Programming), и я рекомендую вам основательно познакомиться с этими технологиями.

Где найти дополнительную информацию

Книги по процессам разработки программного обеспечения всегда были распространены, а расцвет гибкой разработки привел к появлению множества новых публикаций. Моим любимым изданием по основным аспектам процесса является книга [31]. В ней с практической точки зрения рассмотрены многие моменты процесса разработки программных продуктов и приведен большой перечень полезных приемов.

Хороший обзор дается в книгах представителей сообщества, поддерживающего гибкие процессы, [9] и [22]. Много хороших советов по применению UML в режиме гибкой разработки можно найти в [1].

Одним из наиболее популярных методов гибкой разработки является XP (Extreme Programming – экстремальное программирование), в изучении которого помогут веб-сайты *http://xprogramming.com* и *http://www.extremeprogramming.org*. По XP написано множество книг, вот почему теперь я называю его бывшим легковесным методом. Обычно изучение начинают с книги [2].

Хотя книга [3] посвящена XP, в ней подробно рассказывается о планировании итеративных проектов. Большую часть этой информации можно найти и в других книгах по XP, но если вас интересует только аспект планирования, то она будет хорошим выбором.

Дополнительную информацию по Rational Unified Process вы найдете в книге [28] – моем любимом введении в данный вопрос.

3

Диаграммы классов: основы

Если кто-нибудь подойдет к вам в темном переулке и спросит: «Хотите посмотреть на диаграмму UML?», знайте – скорее всего, речь идет о диаграмме класса. Большинство диаграмм UML, которые я встречал, были диаграммами классов. Помимо своего широкого применения диаграммы классов концентрируют в себе большой диапазон понятий моделирования. Хотя их основные элементы используются практически всеми, более сложные элементы применяются не так часто. Именно поэтому я разделил рассмотрение диаграмм классов на две части: основы (данная глава) и дополнительные понятия (глава 5).

Диаграмма классов описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами. В UML термин **функциональность** (feature) применяется в качестве основного термина, описывающего и свойства, и операции класса.

На рис. 3.1 изображена типичная модель класса, понятная каждому, кто имел дело с обработкой заказов клиентов. Прямоугольники на диаграмме представляют классы и разделены на три части: имя класса (жирный шрифт), его атрибуты и его операции. На рис. 3.1 также показаны два вида связей между классами: ассоциации и обобщения.

Свойства

Свойства представляют структурную функциональность класса. В первом приближении можно рассматривать свойства как поля класса. Как мы увидим позднее, в действительности это не так просто, но вполне приемлемо для начала.

Свойства представляют единое понятие, воплощающееся в двух совершенно различных сущностях: в атрибутах и в ассоциациях. Хотя на диаграмме они выглядят совершенно по-разному, в действительности это одно и то же.

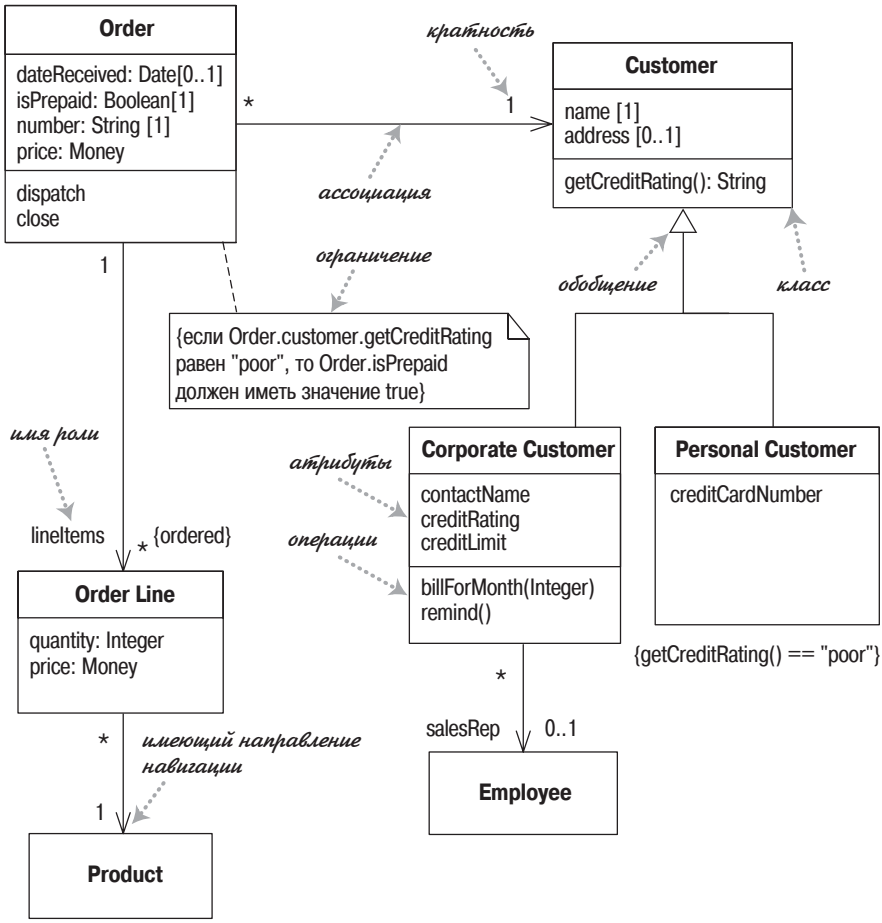

Рис. 3.1. *Простая диаграмма класса*

Атрибуты

Атрибут описывает свойство в виде строки текста внутри прямоугольника класса. Полная форма атрибута:

```
видимость имя: тип кратность = значение по умолчанию {строка свойств}
```

Например:

```
- имя: String [1] = "Без имени" {readOnly}
```

Обязательно только имя.

- Метка `видимость` обозначает, относится ли атрибут к открытым (+) (public) или к закрытым (-) (private). Другие типы видимости обсуждаются на *стр. 110*.

- Имя атрибута – способ ссылки класса на атрибут – приблизительно соответствует имени поля в языке программирования.
- Тип атрибута накладывает ограничение на вид объекта, который может быть размещен в атрибуте. Можно считать его аналогом типа поля в языке программирования.
- Кратность рассмотрена на *стр. 65*.
- Значение по умолчанию представляет собой значение для вновь создаваемых объектов, если атрибут не определен в процессе создания.
- Элемент {строка свойств} позволяет указывать дополнительные свойства атрибута. В примере он равен {readOnly}, то есть клиенты не могут изменять атрибут. Если он пропущен, то, как правило, атрибут можно модифицировать. Остальные строки свойств будут описаны позже.

Ассоциации

Другая ипостась свойства – это ассоциация. Значительная часть информации, которую можно указать в атрибуте, появляется в ассоциации. На рис. 3.2 и 3.3 показаны одни и те же свойства, представленные в различных обозначениях.

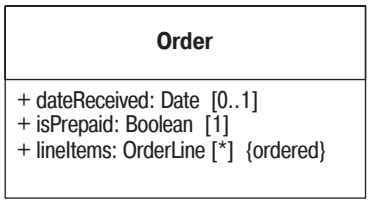

Рис. 3.2. *Представление свойств заказа в виде атрибутов*

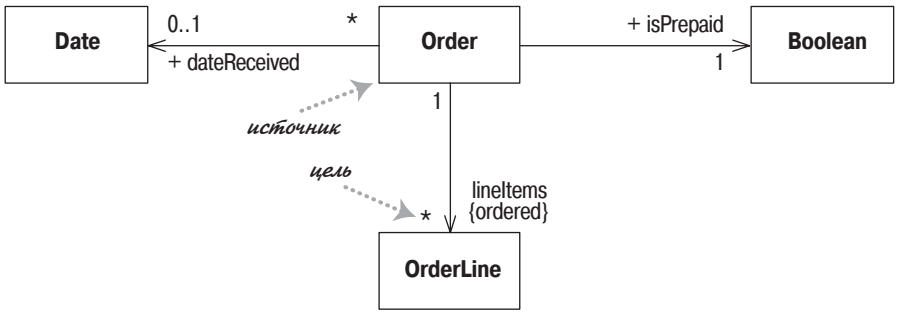

Рис. 3.3. *Представление свойств заказа в виде ассоциаций*

Ассоциация – это непрерывная линия между двумя классами, направленная от исходного класса к целевому классу. Имя свойства (вместе с кратностью) располагается на целевом конце ассоциации. Целевой

конец ассоциации указывает на класс, который является типом свойства. Большая часть информации в обоих представлениях одинакова, но некоторые элементы отличаются друг от друга. В частности, ассоциация может показывать кратность на обоих концах линии.

Естественно, возникает вопрос: «Когда следует выбирать то или иное представление?». Как правило, я стараюсь обозначать при помощи атрибутов небольшие элементы, такие как даты или логические значения, – главным образом, типы значений (*стр. 101*), – а ассоциации для более значимых классов, таких как клиенты или заказы. Я также предпочитаю использовать прямоугольники классов для наиболее значимых классов диаграммы, а ассоциации и атрибуты для менее важных элементов этой диаграммы.

Кратность

Кратность свойства обозначает количество объектов, которые могут заполнять данное свойство. Чаще всего встречаются следующие кратности:

- 1 (Заказ может представить только один клиент.)
- 0..1 (Корпоративный клиент может иметь, а может и не иметь единственного торгового представителя.)
- * (Клиент не обязан размещать заказ, и количество заказов не ограничено. Он может разместить ноль или более заказов.)

В большинстве случаев кратности определяются своими нижней и верхней границами, например 2..4 для игроков в канасту. Нижняя граница может быть нулем или положительным числом, верхняя граница представляет собой положительное число или * (без ограничений). Если нижняя и верхняя границы совпадают, то можно указать одно число; поэтому 1 эквивалентно 1..1. Поскольку это общий случай, * является сокращением 0..*.

При рассмотрении атрибутов вам могут встретиться термины, имеющие отношение к кратности.

- **Optional** (необязательный) предполагает нулевую нижнюю границу.
- **Mandatory** (обязательный) подразумевает, что нижняя граница равна или больше 1.
- **Single-valued** (однозначный) – для такого атрибута верхняя граница равна 1.
- **Multivalued** (многозначный) имеет в виду, что верхняя граница больше 1; обычно *.

Если свойство может иметь несколько значений, я предпочитаю употреблять множественную форму его имени. По умолчанию элементы с множественной кратностью образуют множество, поэтому если вы просите клиента разместить заказы, то они приходят не в произволь-

ном порядке. Если порядок заказов в ассоциации имеет значение, то в конце ассоциации необходимо добавить {ordered}. Если вы хотите разрешить повторы, то добавьте {nonunique}. (Если желательно явным образом показать значение по умолчанию, то можно использовать {unordered} и {unique}.) Встречаются также имена для unordered, nonunique, ориентированные на коллекции, такие как {bag}.

UML 1 допускал дискретные кратности, например 2,4 (означающую 2 или 4, как в случае автомобилей, до того как появились минивэны). Дискретные кратности не были широко распространены, и в UML 2 их уже нет.

Кратность атрибута по умолчанию равна [1]. Хотя это и верно для метамодели, нельзя предполагать, что если значение кратности для атрибута на диаграмме опущено, то оно равно [1], поскольку информация о кратности на диаграмме может отсутствовать. Поэтому я предпочитаю указывать кратность явным образом, если эта информация важна.

Программная интерпретация свойств

Как и для других элементов UML, интерпретировать свойства в программе можно по-разному. Наиболее распространенным представлением является поле или свойство языка программирования. Так, класс Order Line (Строка заказа), показанный на рис. 3.1, мог бы быть представлен в Java следующим образом:

```
public class OrderLine...
  private int quantity;
  private Money price;
  private Order order;
  private Product product
```

В языке, подобном C#, который допускает свойства, это могло бы выглядеть так:

```
public class OrderLine ...
  public int Quantity;
  public Money Price;
  public Order Order;
  public Product Product;
```

Обратите внимание, что атрибут обычно соответствует открытым (public) свойствам в языке, поддерживающем свойства, но соответствует закрытым (private) полям в языке, в котором такой поддержки нет. В языке без свойств с полями можно общаться посредством методов доступа (получение и установка). У атрибута только для чтения не будет метода установки (в случае полей) или операции установки (в случае свойства). Учтите, что если свойству не присвоить имя, то в общем случае ему будет назначено имя целевого класса.

Применение закрытых полей является интерпретацией, ориентированной сугубо на реализацию. Интерпретация, ориентированная в большей степени на интерфейс, может быть акцентирована на методах доступа, а не на данных. В этом случае атрибуты класса `Order Line` могли бы быть представлены следующими методами:

```
public class OrderLine...
  private int quantity;
  private Product product;
  public int getQuantity() {
    return quantity;
  }
  public void setQuantity(int quantity) {
    this.quantity = quantity;
  }
  public Money getPrice() {
    return product.getPrice().multiply(quantity);
  }
```

Здесь нет поля для цены – ее значение вычисляется. Но поскольку клиенты класса `Order Line` заинтересованы в этой информации, она выглядит как поле. Клиенты не могут сказать, что является полем, а что вычисляется. Такое сокрытие информации представляет сущность инкапсуляции.

Если атрибут имеет несколько значений, то связанные с ним данные представляют собой коллекцию. Поэтому класс `Order` (Заказ) будет ссылаться на коллекцию классов `Order Line`. Поскольку эта кратность упорядочена (ordered), то и коллекция должна быть упорядочена (например, List в Java или IList в .NET). Если коллекция не упорядочена, то, строго говоря, она не должна иметь ярко выраженного порядка, то есть должна быть представлена множеством, но большинство специалистов реализуют неупорядоченные атрибуты также в виде списков. Некоторые разработчики используют массивы, но поскольку UML подразумевает неограниченность сверху, то я почти всегда для структуры данных применяю коллекцию.

Многозначные свойства имеют интерфейс, отличный от интерфейса свойств с одним значением (в Java):

```
class Order {
  private Set lineItems = new HashSet();
  public Set getLineItems() {
    return Collections.unmodifiableSet(lineItems);
  }
  public void addLineItem (OrderItem arg) {
    lineItems.add (arg);
  }
  public void removeLineItem (OrderItem arg) {
    lineItems.remove(arg);
  }
```

В большинстве случаев значения многозначных свойств не присваиваются прямо; вместо этого применяются методы добавления (add) и удаления (remove). Для того чтобы управлять своим свойством `Line Items` (Позиции заказов), заказ должен контролировать членство этой коллекции; поэтому он не должен передавать незащищенную коллекцию. В таких случаях я использовал представителя защиты, чтобы заключить коллекцию в оболочку только для чтения. Можно также реализовать необновляемый итератор или сделать копию. Конечно, так клиентам удобнее модифицировать объекты-члены, но они не должны иметь возможность напрямую изменять саму коллекцию.

Поскольку многозначные атрибуты подразумевают коллекции, то практически вы никогда не увидите классы коллекций на диаграмме класса. Их можно увидеть только на очень низком уровне представления диаграмм самих коллекций.

Необходимо крайне остерегаться классов, являющихся не чем иным, как коллекциями полей и средствами доступа к ним. Объектно-ориентированное проектирование должно предоставлять объекты с богатым поведением, поэтому они не должны просто обеспечивать данными другие объекты. Если данные запрашиваются многократно с помощью средств доступа, то это сигнал к тому, что такое поведение должно быть перенесено в объект, владеющий этими данными.

Эти примеры также подтверждают тот факт, что между UML и программой нет обязательного соответствия, однако есть подобие. Соглашения, принятые внутри команды разработчиков, приведут к более полному соответствию.

Независимо от того, как реализовано свойство – в виде поля или как вычисляемое значение, оно представляет нечто, что объект может всегда предоставить. Не следует прибегать к свойству для моделирования транзитного отношения, такого, когда объект передается в качестве параметра во время вызова метода и используется только в рамках данного взаимодействия.

Двунаправленные ассоциации

До сих пор мы говорили об однонаправленных ассоциациях. К другому распространенному типу ассоциаций относится двунаправленная ассоциация, например, показанная на рис. 3.4.

Двунаправленная ассоциация – это пара свойств, связанных в противоположных направлениях. Класс `Car` (Автомобиль) имеет свойство

Рис. 3.4. *Двунаправленная ассоциация*

owner:Person[1], а класс Person (Личность) имеет свойство cars:Car[*]. (Обратите внимание, что я использовал множественную форму имени свойства cars, а это соглашение общепринятое, но ненормативное.)

Обратная связь между ними подразумевает, что если вы следуете обоим свойствам, то должны вернуться обратно к множеству, содержащему вашу исходную точку. Например, если я начинаю с конкретной модели MG Midget, нахожу ее владельца, а затем смотрю на множество принадлежащих ему машин, то оно должно включать модель Midget, с которой я начал.

В качестве альтернативы маркировки ассоциации по свойству многие люди, особенно если они имеют опыт моделирования данных, любят именовать ассоциации с помощью глаголов (рис. 3.5), чтобы отношение можно было использовать в предложении. Это вполне допустимо, и можно добавить к ассоциации стрелку, чтобы избежать неопределенности. Большинство разработчиков объектов предпочитают использовать имя свойства, так как оно больше соответствует функциональным назначениям и операциям.

Некоторые разработчики тем или иным способом именуют каждую ассоциацию. Я предпочитаю давать имя ассоциации, только если это улучшает понимание. Слишком часто встречаются такие имена, как «has» (имеет) или «is related to» (связан с).

На рис. 3.4 двунаправленная природа ассоциации подчеркивается стрелками на обоих концах ассоциации. На рис. 3.5 стрелок нет; в языке UML эта форма применяется либо для обозначения двунаправленной ассоциации, либо когда направление отношения не показывается. Я предпочитаю обозначать двунаправленную ассоциацию с помощью двойных стрелок.

Рис. 3.5. *Использование глагола (own – владеть) в имени ассоциации*

Реализация двунаправленной ассоциации в языке программирования часто представляет некоторую сложность, поскольку необходимо обеспечить синхронизацию обоих свойств. В C# для реализации двунаправленной ассоциации я делаю следующее:

```
class Car...
  public Person Owner {
    get {return __owner;}
    set {
      if (_owner != null) __owner.friendCars().Remove(this);
      __owner = value;
      if (_owner != null)__owner.friendCars().Add(this);
```

```
    }
  }
  private Person _owner;
...
class Person ...
  public IList Cars {
    get {return ArrayList.ReadOnly(_cars);}
  }
  public void AddCar(Car arg) {
    arg.Owner = this;
  }
  private IList _cars = new ArrayList();
  internal IList friendCars() {
    //должен быть использован только Car.Owner
    return _cars;
  }
....
```

Главное – сделать так, чтобы одна сторона ассоциации (по возможности с единственным значением) управляла всем отношением. Для этого ведомый конец (Person) должен предоставить инкапсуляцию своих данных ведущему концу. Это приводит к добавлению в ведомый класс не очень удобного метода, которого здесь не должно было бы быть в действительности, если только язык не имеет более тонкого инструмента управления доступом. Я здесь употребил слово «friend» (друг) в имени как намек на C++, где метод установки ведущего класса действительно был бы дружественным. Как и большинство кода, работающего со свойствами, это стереотипный фрагмент, и поэтому многие разработчики предпочитают получать его посредством различных способов генерации кода.

В концептуальных моделях навигация не очень важна, поэтому в таких случаях я не показываю каких-либо навигационных стрелок.

Операции

Операции (operations) представляют собой действия, реализуемые некоторым классом. Существует очевидное соответствие между операциями и методами класса. Обычно можно не показывать такие операции, которые просто манипулируют свойствами, поскольку они и так подразумеваются.

Полный синтаксис операций в языке UML выглядит следующим образом:

```
видимость имя (список параметров) : возвращаемый тип {строка свойств}
```

- Метка `видимости` обозначает, относится ли операция к открытым (+) (public) или к закрытым (-) (private); другие типы видимости обсуждаются на *стр. 110*.

- `Имя` – это строка.
- `Список параметров` – список параметров операции.
- `Возвращаемый тип` – тип возвращаемого значения, если таковое есть.
- `Строка свойств` – значения свойств, которые применяются к данной операции.

Параметры в списке параметров обозначаются таким же образом, что и для атрибутов. Они имеют вид:

```
направление имя: тип = значение по умолчанию
```

- Имя, тип и значение по умолчанию те же самые, что и для атрибутов.
- Направление обозначает, является ли параметр входным (`in`), выходным (`out`) или тем и другим (`inout`). Если направление не указано, то предполагается `in`.

Например, в счете операция может выглядеть так:

```
+ balanceOn (date: Date) : Money
```

В рамках концептуальной модели не следует применять операции для спецификации интерфейса класса. Вместо этого используйте их для представления главных обязанностей класса, возможно, с помощью пары слов, обобщающих ответственность в CRC-карточках (*стр. 89*).

По моему мнению, следует различать операции, изменяющие состояние системы, и операции, не делающие этого. Язык UML определяет **запрос** как некую операцию, результатом которой является некоторое значение, получаемое от класса; при этом состояние системы не изменяется, то есть данная операция не вызывает побочных эффектов. Такую операцию можно пометить строкой свойств `{query}` (запрос). Операции, изменяющие состояние, я называю **модификаторами**, иначе именуемые командами.

Строго говоря, различие между запросом и модификаторами состоит в том, могут ли они изменять видимое состояние [33]. Видимое состояние – это то, что можно наблюдать извне. Операция, обновляющая кэш, изменит внутреннее состояние, но это не окажет никакого влияния на то, что видно снаружи.

Я считаю полезным выделение запросов, так как это позволяет изменить порядок выполнения запросов и не изменить при этом поведение системы. Общепринято конструировать операции так, чтобы модификаторы не возвращали значение, – тогда можно быть уверенным в том, что операции, возвращающие значения, являются запросами. [33] называет это принципом разделения команды-запроса. Делать так все время не очень удобно, но необходимо применять этот способ так часто, как только возможно.

Другие термины, с которыми иногда приходится сталкиваться, – это методы получения значения (getting methods) и методы установки

значения (setting methods). **Метод получения 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 модификатор – методом установки значений. Эта информация о методах является исключительно внутренней для каждого из классов.

Существует еще различие между операцией и методом. **Операция** представляет собой то, что вызывается объектом – объявление процедуры, тогда как **метод** – это тело 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* употребляются как взаимозаменяемые, однако иногда полезно их различать.

Обобщение

Типичный пример **обобщения** (generalization) включает индивидуального и корпоративного клиентов некоторой бизнес-системы. Несмотря на определенные различия, у них много общего. Одинаковые свойства можно поместить в базовый класс `Customer` (Клиент, супертип), при этом класс `Personal Customer` (Индивидуальный клиент) и класс `Corporate Customer` (Корпоративный клиент) будут выступать как подтипы.

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

С точки зрения программного обеспечения очевидная интерпретация наследования выглядит следующим образом: Корпоративный клиент является подклассом класса Клиент. В основных объектно-ориентированных языках подкласс наследует всю функциональность суперкласса и может переопределять любые методы суперкласса.

Важным принципом эффективного использования наследования является **замещаемость**. Мне необходимо иметь возможность подставить Корпоративного клиента в любом месте программы, где требуется Клиент, и при этом все должно прекрасно работать. По существу это означает, что когда я пишу программу в предположении, что у меня есть Клиент, то я могу свободно использовать любой подтип Клиента.

Вследствие полиморфизма Корпоративный клиент может реагировать на определенные команды не так, как другой Клиент, но вызывающий не должен беспокоиться об этом отличии. (Дополнительную информацию можно найти в главе «Liskov Substitution Principle (LSP)» (Принцип замещения Лисков) книги [30]).

Наследование представляет собой мощный механизм, но оно несет с собой много такого, что не всегда является необходимым для достижения замещаемости. Вот хороший пример: на заре существования языка Java многим разработчикам не нравилась реализация встроенного класса `Vector` (Вектор), и они хотели заменить его чем-нибудь полегче. Однако единственным способом получения класса, способного заменить `Vector`, было создание его подкласса, что означало наследование множества нежелательных данных и поведения.

Замещаемые классы можно создавать при помощи массы других механизмов. Поэтому многие разработчики предпочитают различать создание подтипа, то есть наследование интерфейса, и создание подкласса, или наследование реализации. Класс – это **подтип**, если он может замещать свой супертип, в независимости от того, использует он наследование или нет. Создание **подкласса** используется как синоним обычного наследования.

Существует достаточное количество других механизмов, позволяющих создавать подтипы без создания подклассов. Примером может служить реализация интерфейса (*стр. 96*) и множество стандартных шаблонов разработки [21].

Примечания и комментарии

Примечания – это комментарии на диаграммах. Примечания могут существовать сами по себе или быть связаны пунктирной линией с элементами, которые они комментируют (рис. 3.6). Они могут присутствовать на диаграммах любого типа.

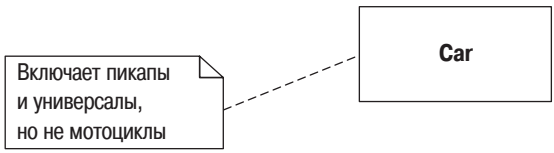

Рис. 3.6. *Примечание используется как комментарий к одному или более элементам диаграммы*

Иногда применять пунктирную линию неудобно из-за невозможности точного позиционирования конца линии. Поэтому по общепринятому соглашению в конце линии размещается небольшая открытая окружность. В некоторых случаях удобнее поместить однострочный комментарий на элементе диаграммы, при этом в начале текста ставятся два дефиса: --.

Зависимость

Считается, что между двумя элементами существует **зависимость** (dependency), если изменения в определении одного элемента **(сервера)** могут вызвать изменения в другом элементе **(клиенте)**. В случае классов зависимости появляются по разным причинам: один класс посылает сообщение другому классу; один класс владеет другим классом как частью своих данных; один класс использует другой класс в качестве параметра операции. Если класс изменяет свой интерфейс, то сообщения, посылаемые этому классу, могут стать недействительными.

По мере роста систем необходимо все более и более беспокоиться об управлении зависимостями. Если зависимости выходят из-под контроля, то каждое изменение в системе оказывает действие, нарастающее волнообразно по мере увеличения количества изменений. Чем больше волна, тем труднее что-нибудь изменить.

UML позволяет изобразить зависимости между элементами всех типов. Зависимости можно использовать всякий раз, когда надо показать, как изменения в одном элементе могут повлиять на другие элементы.

На рис. 3.7 показаны зависимости, которые можно обнаружить в многоуровневом приложении. Класс `Benefits Window` (Окно льгот) – это пользовательский интерфейс, или класс **представления**, зависящий от класса `Employee` (Сотрудник). Класс `Employee` – это **объект предметной области**, который представляет основное поведение системы, в данном случае бизнес-правила. Это означает, что если класс `Employee` изменяет свой интерфейс, то, возможно, и класс `Benefits Window` также должен измениться.

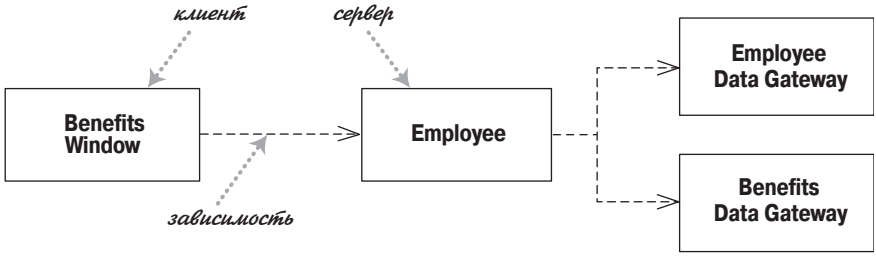

Рис. 3.7. *Пример зависимостей*

Здесь важно то, что зависимость имеет только одно направление и идет от класса представления к классу предметной области. Таким образом, мы знаем, что имеем возможность свободно изменять класс `Benefits Window`, не оказывая влияния на объект `Employee` или другие объекты предметной области. Я понял, что строгое разделение логики представления и логики предметной области, когда представление зависит от предметной области, но не наоборот, – это ценное правило, которому я должен следовать.

Второй существенный момент этой диаграммы: здесь нет прямой зависимости двух классов Data Gateway (Шлюз данных) от Benefits Window. Если эти классы изменяются, то, возможно, должен измениться и класс Employee. Но если изменяется только реализация класса Employee, а не его интерфейс, то на этом изменения и заканчиваются.

UML включает множество видов зависимостей, каждая с определенной семантикой и ключевыми словами. Базовая зависимость, которую я здесь обрисовал, с моей точки зрения, наиболее полезна, и обычно я использую ее без ключевых слов. Чтобы сделать ее более детальной, вы можете добавить соответствующее ключевое слово (табл. 3.1).

Таблица 3.1. Избранные ключевые слова зависимостей

Ключевое слово	Значение
«call» (вызывать)	Источник вызывает операцию в цели
«create» (создавать)	Источник создает экземпляр цели
«derive» (производить)	Источник представляет собой производное цели
«instantiate» (создать экземпляр)	Источник является экземпляром цели. (Обратите внимание, что если источник является классом, то сам класс является экземпляром класса класс; то есть целевой класс – это метакласс)
«permit» (разрешать)	Цель разрешает источнику доступ к ее закрытой функциональности
«realize» (реализовать)	Источник является реализацией спецификации или интерфейса, определенного целью (*стр. 96*)
«refine» (уточнить)	Уточнение означает отношение между различными семантическими уровнями; например, источник может быть классом разработки, а цель – соответствующим классом анализа
«substitute» (заместить)	Источник может быть заменен целью (*стр. 72*)
«trace» (проследить)	Используется, чтобы отследить такие моменты, как требования к классам или как изменения одной ссылки модели влияют на все остальное
«use» (использовать)	Для реализации источника требуется цель

Базовая зависимость не является транзитивным отношением. Примером **транзитивного** отношения может служить отношение «эта борода больше». Если у Джима борода больше, чем у Гради, а борода Гради больше бороды Айвара, то мы можем сделать вывод, что у Джима борода больше, чем у Айвара. Некоторые типы зависимостей, такие как замещение, являются транзитивными, но в большинстве случаев существует значительное расхождение между прямыми и обратными зависимостями, как показано на рис. 3.7.

Многие отношения UML предполагают зависимость. Направленная ассоциация от Order к Customer означает, что Order зависит от Customer. Подкласс зависит от своего суперкласса, но не наоборот.

Вашим основным правилом должна стать минимизация зависимостей, особенно когда они затрагивают значительную часть системы. В частности, будьте осторожны с циклами, поскольку они могут привести к циклическим изменениям. Я не слишком строго придерживаюсь этого правила. Я не имею в виду взаимные зависимости между тесно связанными классами, но стараюсь избегать циклов на более высоких уровнях, особенно между пакетами.

Бесполезно пытаться показать все зависимости на диаграмме классов; их слишком много, и они слишком сильно отличаются. Соблюдайте меру и показывайте только зависимости, относящиеся к конкретной теме, о которой вы хотите сообщить. Чтобы понимать и управлять зависимостями, лучше всего использовать для этого диаграммы пакетов (*стр. 114*).

В самом общем случае я использую зависимости с классами для иллюстрации транзитивного отношения, такого, когда один объект передается другому в качестве параметра. Иногда их применяют с ключевыми словами «parameter» (параметр), «local» (локальный) и «global» (глобальный). Эти ключевые слова можно увидеть на ассоциациях в моделях UML 1, и в этом случае они обозначают транзитные связи, а не свойства. Эти ключевые слова не входят в UML 2.

Зависимости можно обнаружить, просматривая программу, вот почему эти инструменты идеальны для анализа зависимостей. Применение инструментария для обращения схем зависимостей – наиболее полезный способ применения этого раздела UML.

Правила ограничений

При построении диаграмм классов большая часть времени уходит на представление различных ограничений. На рис. 3.1 показано, что Заказ (Order) может быть сделан только одним единственным Клиентом (Customer). Из этой диаграммы классов также следует, что каждая Line Item (Позиция заказа) рассматривается отдельно: вы можете заказать 40 коричневых, 40 голубых и 40 красных штучек, но не 120 штук вообще. Далее диаграмма утверждает, что Корпоративные клиенты располагают кредитами, а Индивидуальные клиенты – нет.

С помощью базовых конструкций ассоциации, атрибута и обобщения можно сделать многое, 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 описания ограничений все что угодно. При этом необходимо лишь придерживаться правила: ограничения следует помещать в фигурные скобки ({}). Можно употреблять разговорный язык, язык программирования или формальный объектный язык ограничений UML (Object Constraint Language, OCL) [43], базирующийся на исчислении предикатов. Формальное написание позволяет избежать риска неоднозначного толкования конструкций разговорного языка. Однако это приводит к возможности недоразумений из-за непрофессионального владения OCL пишущими и читающими. Поэтому до тех пор, пока ваши читатели не вполне овладеют исчислением предикатов, я предлагаю говорить на обычном языке.

Если хотите, можете предварять ограничение именем с двоеточием, например: {запрещение кровосмешения: муж и жена не должны быть родными братом и сестрой}.

Когда применяются диаграммы классов

Диаграммы классов составляют фундамент UML, и поэтому их постоянное применение является условием достижения успеха. Эта глава посвящена основным понятиям, а многие более сложные материи обсуждаются в главе 5.

Трудность, связанная с диаграммами классов, заключается в том, что они настолько обширны, что их применение может оказаться непомерно сложным. Приведем несколько полезных советов.

- Не пытайтесь задействовать сразу все доступные понятия. Начните с самых простых, описанных в этой главе: классов, ассоциаций, атрибутов, обобщений и ограничений. Обращайтесь к дополнительным понятиям, рассмотренным в главе 5, только если они действительно необходимы.
- Я пришел к выводу, что концептуальные диаграммы классов очень полезны при изучении делового языка. Чтобы при этом все получалось, необходимо всячески избегать обсуждения программного обеспечения и применять очень простые обозначения.
- Не надо строить модели для всего на свете, вместо этого следует сконцентрироваться на ключевых аспектах. Лучше создать мало диаграмм, которые постоянно применяются 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 сосредоточиться исключительно на структуре и забыть о поведении. Поэтому, рисуя диаграммы классов для того, чтобы разобраться в программном обеспечении, используйте какие-либо формы анализа поведения. Если вы применяете эти методы поочередно, значит, вы двигаетесь в верном направлении.

Проектирование по контракту

Проектирование по контракту (Design by Contract) – это метод проектирования, являющийся центральным свойством языка Eiffel. И метод, и язык разработаны Бертраном Мейером [33]. Однако проектирование по контракту не является привилегией только языка Eiffel, этот метод **можно применять** и в любом другом языке программирования.

Главной идеей проектирования по контракту является понятие утверждения. **Утверждение** (assertion) – это булево высказывание, которое никогда не должно принимать ложное значение и поэтому может быть ложным только в результате ошибки. Обычно утверждение проверяется только во время отладки и не проверяется в режиме выполнения. Действительно при выполнении программы никогда не следует предполагать, что утверждение проверяется.

В методе проектирования по контракту определены утверждения трех типов: 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 *input = result * result*, где *result* является выходом, а *input* – 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 *input >= 0*. Такое предусловие утверждает, что применение операции «извлечь квадратный корень» для отрицательного числа является ошибочным и последствия такого применения не определены.

На первый взгляд эта идея кажется неудачной, поскольку нам придется выполнить некоторые дополнительные проверки, чтобы убедиться в корректности выполнения операции «извлечь квадратный корень». При этом возникает важный вопрос: на кого ляжет ответственность за выполнение этой проверки.

Предусловие явным образом устанавливает, что за подобную проверку отвечает вызывающий объект. Без такого явного указания обязанностей мы можем получить либо недостаточный уровень проверки (когда каждая из сторон предполагает, что ответственность несет другая сторона), либо чрезмерную проверку (когда она будет выполняться обеими сторонами). Излишняя проверка тоже плоха, поскольку это влечет за собой дублирование кода проверки, что, в свою очередь, может существенно увеличить сложность про-

граммы. Явное определение ответственности помогает снизить сложность кода. Опасность того, что вызывающий объект забудет выполнить проверку, уменьшается тем обстоятельством, что утверждение обычно проверяется во время отладки и тестирования.

Исходя из этих определений предусловия и постусловия, мы можем дать строгое определение термина **исключение**. Исключение возникает, когда предусловие операции выполнено, но операция не может возвратить значение в соответствии с указанным постусловием.

Инвариант представляет собой утверждение относительно класса. Например, класс Account (Счет) может иметь инвариант, который утверждает, что *balance == sum(entries.amount( ))*. Инвариант должен быть «всегда» истинным для всех экземпляров класса. В данном случае «всегда» означает «всякий раз, когда объект доступен для выполнения над ним операции».

По существу это означает, что инвариант дополняет предусловия и постусловия, связанные со всеми открытыми операциями данного класса. Значение инварианта может оказаться ложным во время выполнения некоторого метода, однако оно должно снова стать истинным к моменту взаимодействия с любым другим объектом.

Утверждения могут играть уникальную роль в определении подклассов. Одна из опасностей наследования состоит в том, что операции подкласса можно переопределить так, что они станут не совместимыми с операциями суперкласса. Утверждения уменьшают вероятность этого. Инварианты и постусловия класса должны применяться ко всем подклассам. Подклассы могут усилить эти утверждения, но не могут их ослабить. С другой стороны, предусловия нельзя усилить, но можно ослабить.

На первый взгляд все это кажется излишним, однако имеет весьма важное значение для обеспечения динамического связывания. В соответствии с принципом замещения необходимо всегда иметь возможность обратиться к объекту подкласса так, как если бы он был экземпляром суперкласса. Если подкласс усилил свое предусловие, то операция суперкласса, примененная к подклассу, может завершиться аварийно.

Где найти дополнительную информацию

Все упомянутые мной в главе 1 книги по основам UML рассказывают о диаграммах классов более подробно. Управление зависимостями является критическим элементом больших проектов. Лучшая книга по этой теме – [30].

4

Диаграммы последовательности

Диаграммы взаимодействия (interaction diagrams) описывают взаимодействие групп объектов в различных условиях их поведения. UML определяет диаграммы взаимодействия нескольких типов, из которых наиболее употребительными являются диаграммы последовательности.

Обычно диаграмма последовательности описывает один сценарий. На диаграмме показаны экземпляры объектов и сообщения, которыми обмениваются объекты в рамках одного прецедента (use case).

Для того чтобы начать обсуждение, рассмотрим простой сценарий. Предположим, что у нас есть заказ, и мы собираемся вызвать команду для определения его стоимости. При этом объекту заказа (`Order`) необходимо просмотреть все позиции заказа (`Line Items`) и определить их цены, основанные на правилах построения цены продукции в строке заказа (`Order Line`). Проделав это для всех позиций заказа, объект заказа должен вычислить общую скидку, которая определяется индивидуально для каждого клиента.

На рис. 4.1 приведена диаграмма, представляющая реализацию данного сценария. Диаграммы последовательности показывают взаимодействие, представляя каждого участника вместе с его линией жизни (lifeline), которая идет вертикально вниз и упорядочивает сообщения на странице; сообщения также следует читать сверху вниз.

Одно из преимуществ диаграммы последовательности заключается в том, что мне почти не придется объяснять ее нотацию. Можно видеть, что экземпляр заказа посылает строке заказа сообщения `getQuantity` и `getProduct`. Можно также видеть, как заказ применяет метод к самому себе и как этот метод посылает сообщение `getDiscountInfo` экземпляру клиента.

Однако диаграмма не все показывает так хорошо. Последовательность сообщений `getQuantity`, `getProduct`, `getPricingDetails` и `calculateBasePrice` должна быть реализована для каждой строки заказа, тогда как метод `calculateDiscounts` вызывается лишь однажды. Такое заключение нель-

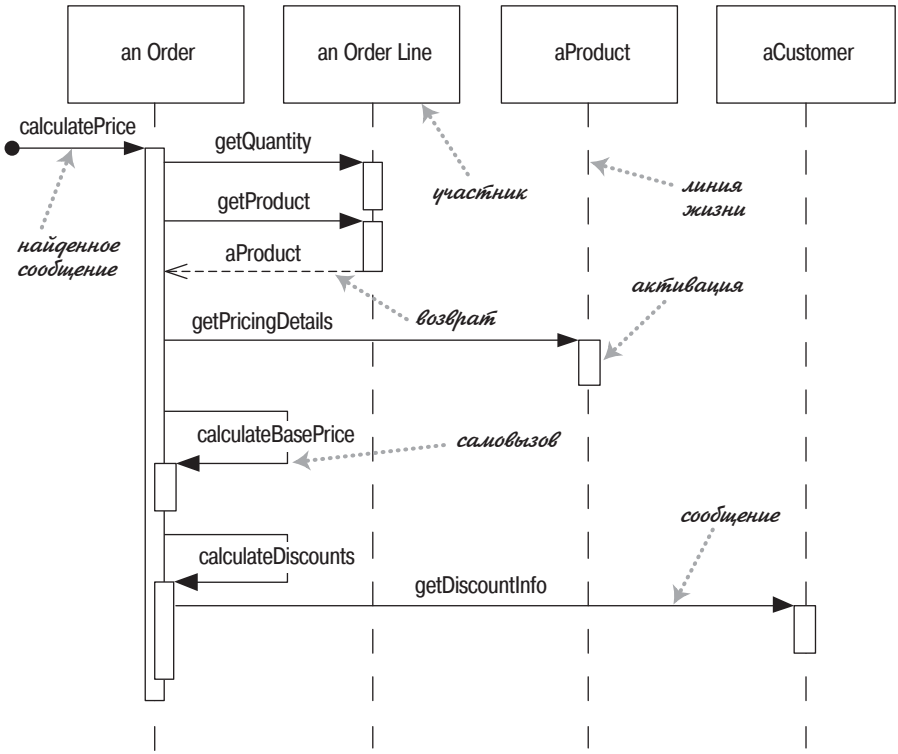

Рис. 4.1. *Диаграмма последовательности централизованного управления*

зя сделать на основе этой диаграммы, но позднее я введу дополнительное обозначение, которое поможет в этом.

В большинстве случаев можно считать участников диаграммы взаимодействия объектами, как это и было в действительности в UML 1. Но в UML 2 их роль значительно сложнее, и полное ее объяснение выходит за рамки этой книги. Поэтому я употребляю термин **участники** (participants), который формально не входит в спецификацию UML. В UML версии 1 участники были объектами, и поэтому их имена подчеркивались, но в UML 2 их надо показывать без подчеркивания, как я и сделал выше.

На приведенной диаграмме я именовал участников, используя стиль `anOrder`. В большинстве случаев это вполне приемлемо. Вот более полный синтаксис: `имя : Класс`, где и имя, и класс не обязательны, но если класс используется, то двоеточие должно присутствовать. (Этот стиль выдержан на рис. 4.4.)

Каждая линия жизни имеет полосу активности, которая показывает интервал активности участника при взаимодействии. Она соответствует времени нахождения в стеке одного из методов участника. В языке UML полосы активности не обязательны, но я считаю их исключи-

тельно удобными при пояснении поведения. Единственным исключением является стадия проработки дизайна, поскольку их неудобно рисовать на белых досках.

Именование бывает часто полезным для установления связей между участниками на диаграмме. Как видно на диаграмме, вызов метода `getProduct` возвращает `aProduct`, имеющего то же самое имя и, следовательно, означающего того же самого участника, `aProduct`, которому посылается вызов `getPricingDetails`. Обратите внимание, что обратной стрелкой я обозначил только этот вызов с целью показать соответствие. Многие разработчики используют возвраты для всех вызовов, но я предпочитаю применять их, только когда это дает дополнительную информацию; в противном случае они просто вносят неразбериху. Не исключено, что даже в данном случае можно было опустить возврат, не запутав читателя.

У первого сообщения нет участника, пославшего его, поскольку оно приходит из неизвестного источника. Оно называется **найденным сообщением** (found message).

Другой подход можно увидеть на рис. 4.2. Основная задача остается той же самой, но способ взаимодействия участников для ее решения совершенно другой. Заказ спрашивает каждую строку заказа о его собственной цене (Price). Сама строка заказа передает вычисление дальше – объекту продукта (Product); обратите внимание, как мы показываем передачу параметра. Подобным же образом для вычисления скидки объект заказа вызывает метод для клиента (Customer). Поскольку для выполнения этой задачи клиенту требуется информация от объекта заказа, то он делает повторный вызов в отношении заказа для получения этих данных.

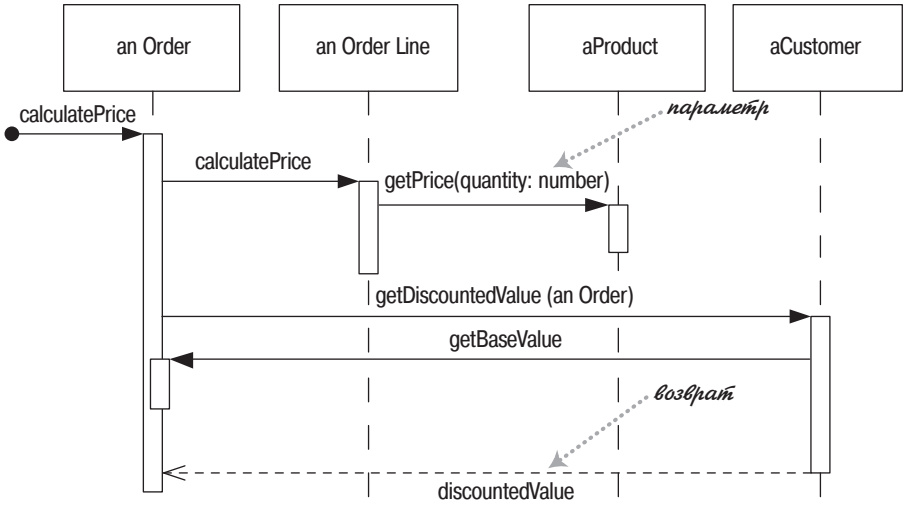

Рис. 4.2. *Диаграмма последовательности для распределенного управления*

Во-первых, на этих двух диаграммах надо обратить внимание на то, насколько ясно диаграмма последовательности показывает различия во взаимодействии участников. В этом проявляется мощь диаграмм взаимодействий. Они не очень хорошо представляют детали алгоритмов, такие как циклы или условное поведение, но делают абсолютно прозрачными вызовы между участниками и дают действительно ясную картину того, какую обработку выполняют конкретные участники.

Во-вторых, посмотрите, как четко видна разница в стиле между двумя взаимодействиями. На рис. 4.1 представлено **централизованное управление** (centralized control), когда один из участников в значительной степени выполняет всю обработку, а другие предоставляют данные. На рис. 4.2 изображено **распределенное управление** (distributed control), при котором обработка распределяется между многими участниками, каждый их которых выполняет небольшую часть алгоритма.

Оба стиля обладают преимуществами и недостатками. Большинство разработчиков, особенно новички в объектно-ориентированном программировании, чаще всего применяют централизованное управление. Во многих случаях это проще, так как вся обработка сосредоточена в одном месте; напротив, в случае распределенного управления при попытке понять программу создается ощущение погони за объектами.

Несмотря на это фанатики объектов, такие как я, предпочитают распределенное управление. Одна из главных задач хорошего проектирования заключается в локализации изменений. Данные и программный код, получающий доступ к этим данным, часто изменяются вместе. Поэтому размещение данных и обращающейся к ним программы в одном месте – первое правило объектно-ориентированного проектирования.

Кроме того, распределенное управление позволяет создать больше возможностей для применения полиморфизма, чем в случае применения условной логики. Если алгоритмы определения цены отличаются для различных типов продуктов, то механизм распределенного управления позволяет нам использовать подклассы класса продукта (Product) для обработки этих вариантов.

Вообще, объектно-ориентированный стиль предназначен для работы с большим количеством небольших объектов, обладающих множеством небольших методов, что дает широкие возможности для переопределения и изменения. Этот стиль сбивает с толку людей, применяющих длинные процедуры; действительно это изменение является сердцем **смены парадигмы** (paradigm shift) при объектной ориентации. Научить этому трудно. Представляется, что единственный способ действительно понять это заключается в использовании распределенного управления при работе в объектно-ориентированном окружении. Многие люди говорят, что они испытали внезапное озарение, когда поняли смысл этого стиля. В этот момент их мозг перестроился, и они начали думать, что децентрализованное управление действительно проще.

Создание и удаление участников

В диаграммах последовательности для создания и удаления участников применяются некоторые дополнительные обозначения (рис. 4.3). В случае создания участника надо нарисовать стрелку сообщения, направленную к прямоугольнику участника. Если применяется конструктор, то имя сообщения не обязательно, но я обычно маркирую его словом «new» в любом случае. Если участник выполняет что-нибудь непосредственно после создания, например команду запроса, то надо начать активацию сразу после прямоугольника участника.

Удаление участника обозначается большим крестом (X). Стрелка сообщения, идущая в X, означает, что один участник явным образом удаляет другого; X в конце линии жизни показывает, что участник удаляет сам себя.

Если в системе работает сборщик мусора, то объекты не удаляются вручную, тем не менее следует при помощи X показать, что объект больше не нужен и готов к удалению. Так следует поступать и в случае операций закрытия, показывая, что объект больше не используется.

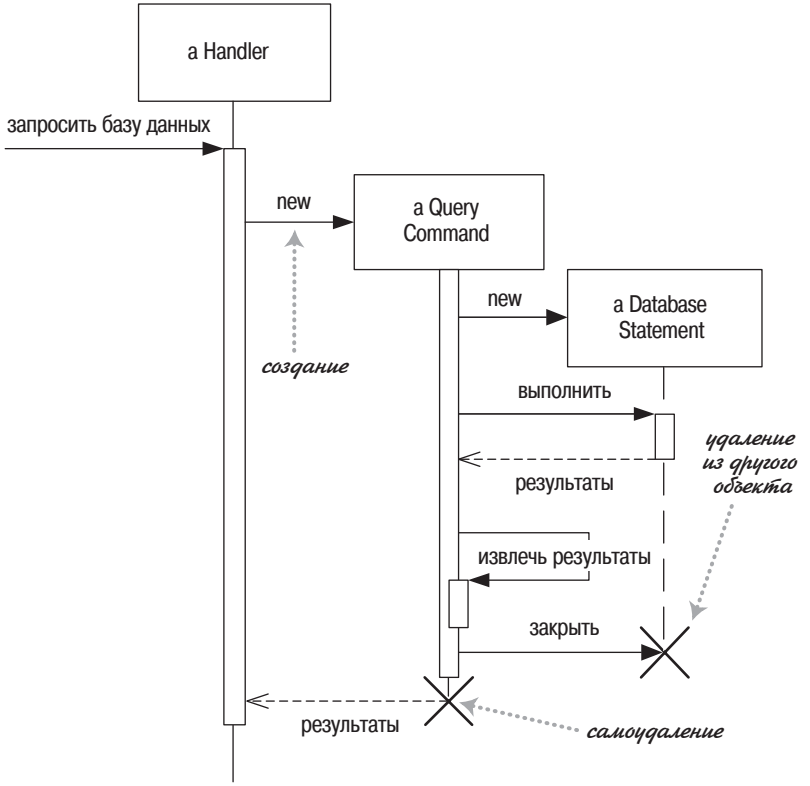

Рис. 4.3. *Создание и удаление участников*

Циклы, условия и тому подобное

Общая проблема диаграмм последовательности заключается в том, как отображать циклы и условные конструкции. Прежде всего надо усвоить, что диаграммы последовательности для этого не предназначены. Подобные управляющие структуры лучше показывать с помощью диаграммы деятельности или собственно кода. Диаграммы последовательности применяются для визуализации процесса взаимодействия объектов, а не как средство моделирования алгоритма управления.

Как было сказано, существуют дополнительные обозначения. И для циклов, и для условий используются **фреймы взаимодействий** (interaction frames), представляющие собой средство разметки диаграммы взаимодействия. На рис. 4.4 показан простой алгоритм, основанный на следующем псевдокоде.

```
foreach (lineitem)
  if (product.value > $10K)
    careful.dispatch
  else
    regular.dispatch
  end if
```

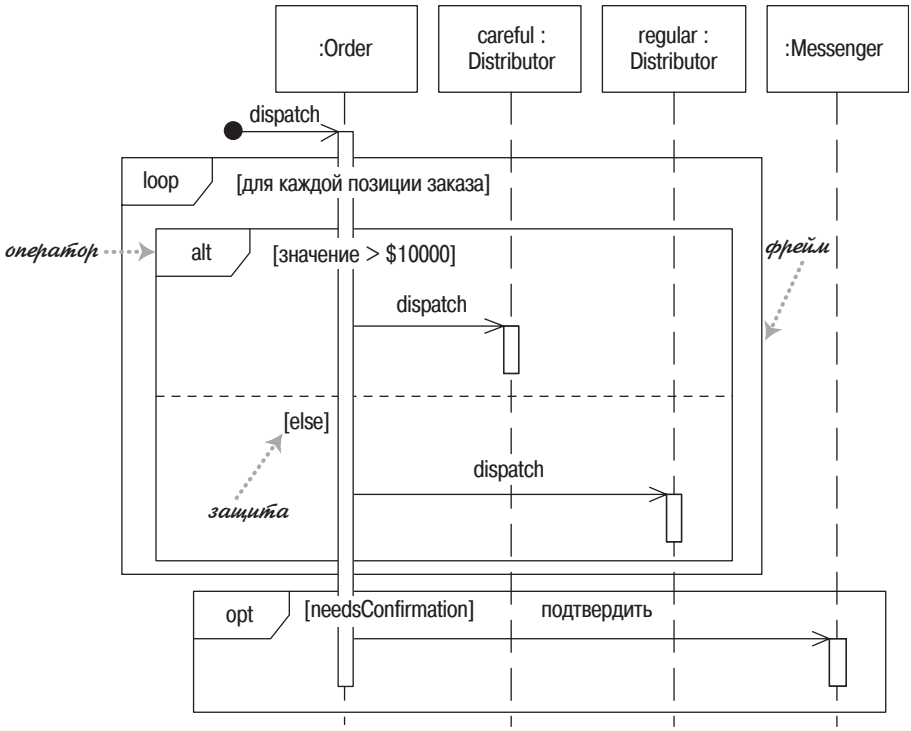

Рис. 4.4. *Фреймы взаимодействия*

```
    end for
    if (needsConfirmation) messenger.confirm
  end procedure
```

В основном фреймы состоят из некоторой области диаграммы последовательности, разделенной на несколько фрагментов. Каждый фрейм имеет оператор, а каждый фрагмент может иметь защиту. (В табл. 4.1 перечислены общепринятые операторы для фреймов взаимодействия.) Для отображения цикла применяется оператор `loop` с единственным фрагментом, а тело итерации помещается в защиту. Для условной логики можно использовать оператор `alt` и помещать условие в каждый фрагмент. Будет выполнен только тот фрагмент, защита которого имеет истинное значение. Для единственной области существует оператор `opt`.

Таблица 4.1. Общепринятые операторы для фреймов взаимодействия

Оператор	Значение
`alt`	Несколько альтернативных фрагментов (alternative); выполняется только тот фрагмент, условие которого истинно (рис. 4.4)
`opt`	Необязательный (optional) фрагмент; выполняется, только если условие истинно. Эквивалентно `alt` с одной веткой (рис. 4.4)
`par`	Параллельный (parallel); все фрагменты выполняются параллельно
`loop`	Цикл (loop); фрагмент может выполняться несколько раз, а защита обозначает тело итерации (рис. 4.4)
`region`	Критическая область (critical region); фрагмент может иметь только один поток, выполняющийся за один прием
`neg`	Отрицательный (negative) фрагмент; обозначает неверное взаимодействие
`ref`	Ссылка (reference); ссылается на взаимодействие, определенное на другой диаграмме. Фрейм рисуется, чтобы охватить линии жизни, вовлеченные во взаимодействие. Можно определять параметры и возвращать значение
`sd`	Диаграмма последовательности (sequence diagram); используется для очерчивания всей диаграммы последовательности, если это необходимо

Фреймы взаимодействия – новинка UML 2. В диаграммах, разработанных до создания UML 2, применяется другой подход; кроме того, некоторые разработчики не любят фреймы и предпочитают прежние соглашения. На рис. 4.5 показаны некоторые из этих неофициальных приемов.

В UML 1 использовались маркеры итераций и защиты. В качестве **маркера итерации** (iteration marker) выступал символ *, добавленный к имени сообщения. Для обозначения тела итерации можно добавить текст в квадратных скобках. **Защита** (guard) – это условное выраже-

ние, размещенное в квадратных скобках и означающее, что сообщение посылается, только когда защита принимает истинное значение. Эти обозначения исключены из UML 2, но они все еще встречаются в диаграммах взаимодействия.

Несмотря на то что маркеры итерации и защиты могут оказаться полезными, они имеют один недостаток. С помощью защиты нельзя показать, что несколько защит взаимно исключают друг друга, например две защиты, представленные на рис. 4.5. Оба обозначения работают только в случае отправки одного сообщения и не работают, когда при одной активации посылается несколько сообщений в рамках того же самого цикла или условного блока.

Решением последней проблемы может служить ставшее популярным неофициальное соглашение, заключающееся в применении **псевдосообщения** (pseudomessage) в виде условия цикла или защиты на одном из вариантов обозначения самовызова. На рис. 4.5 я показал это без стрелки сообщения; некоторые разработчики включают стрелку сообщения, но ее отсутствие помогает подчеркнуть, что это ненастоящий вызов. Некоторые разработчики любят оттенять прямоугольник активации псевдосообщения серым цветом. Вариативное поведение можно показать, поставив маркер альтернативы между активациями.

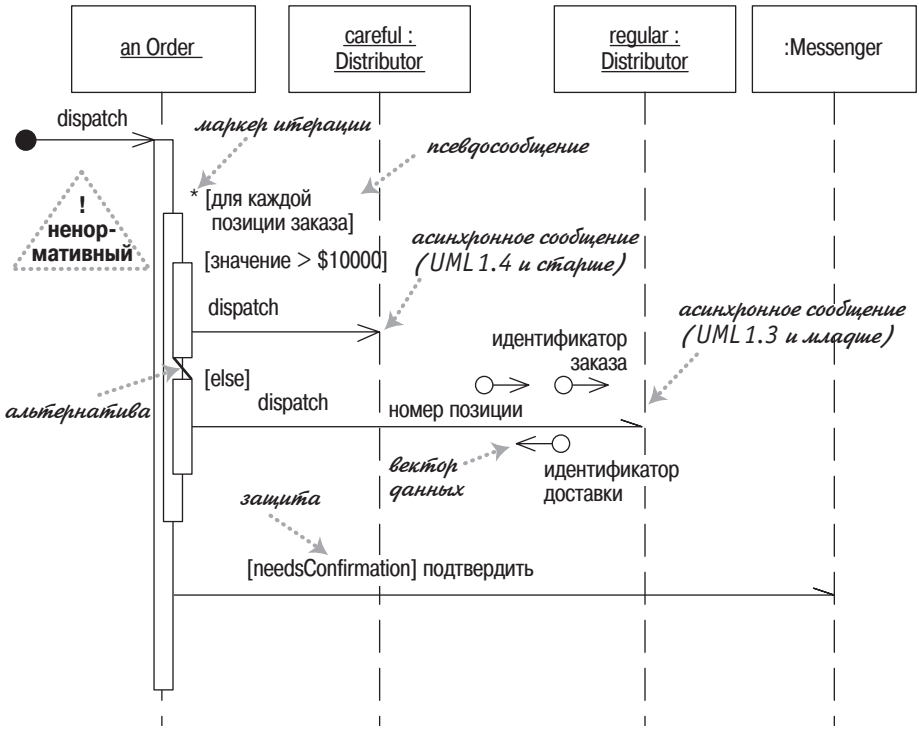

Рис. 4.5. *Старые соглашения для условной логики*

Хотя я считаю активации очень полезными, они не слишком много дают в случае метода dispatch (перенаправить), с помощью которого отправляются сообщения, при этом в рамках активации приемника больше ничего не происходит. По общепринятому соглашению, которого я придерживался в диаграмме на рис. 4.5, для таких простых вызовов активация опускается.

Стандарт UML не предоставляет графических средств для обозначения передаваемых данных; вместо этого они показываются с помощью параметров в имени сообщения и на стрелках возврата. **Векторы данных** повсеместно использовались во многих областях для обозначения перемещения данных, и многие разработчики все еще с удовольствием применяют их в UML.

В конечном счете, хотя различные системы могут включать в диаграммы последовательности обозначения для условной логики, я не думаю, что они работают сколько-нибудь лучше программного кода или, по крайней мере, псевдокода. В частности, я считаю фреймы взаимодействия очень тяжеловесными, скрывающими основной смысл диаграммы, поэтому я предпочитаю псевдосообщения.

Синхронные и асинхронные вызовы

Если вы были очень внимательны, то заметили, что стрелки в последних двух диаграммах отличаются от предыдущих. Это небольшое отличие достаточно важно в UML версии 2. Здесь закрашенные стрелки показывают синхронное сообщение, а простые стрелки обозначают асинхронное сообщение.

Если вызывающий объект посылает **синхронное сообщение** (synchronous message), то он должен ждать, пока обработка сообщения не будет закончена, например при вызове подпрограммы. Если вызывающий объект посылает **асинхронное сообщение** (asynchronous message), то он может продолжать работу и не должен ждать ответа. Асинхронные вызовы можно встретить в многопоточных приложениях и в промежуточном программном обеспечении, ориентированном на сообщения. Асинхронность улучшает способность к реагированию и уменьшает количество временных соединений, но сложнее в отладке.

Разница в изображении стрелок едва уловима; действительно их довольно трудно отличить. Кроме того, это изменение, введенное в UML 1.4, не обладает обратной совместимостью, поскольку до этого асинхронные сообщения обозначались половинными стрелками, как показано на рис. 4.5.

На мой взгляд, такое различие слишком незаметно. Я бы советовал выделять асинхронные сообщения при помощи старых половинных стрелок, которые больше привлекают взгляд. Читая диаграмму последовательности, не спешите делать предположения о синхронности по

виду стрелок до тех пор, пока не убедитесь, что автор умышленно нарисовал их разными.

Когда применяются диаграммы последовательности

Диаграммы последовательности следует применять тогда, когда требуется посмотреть на поведение нескольких объектов в рамках одного прецедента. Диаграммы последовательности хороши для представления взаимодействия объектов, но не очень подходят для точного определения поведения.

Если вы хотите посмотреть на поведение одного объекта в нескольких прецедентах, то примените диаграмму состояния (глава 10). Если же надо изучить поведение нескольких объектов в нескольких прецедентах или потоках, не забудьте о диаграмме деятельности (глава 11).

Если требуется быстро исследовать несколько вариантов взаимодействия, лучше использовать CRC-карточки, поскольку это позволяет избежать непрерывного рисования и стирания. Часто бывает удобно поработать с CRC-карточками для просмотра вариантов взаимодействия, а затем с помощью диаграмм взаимодействий фиксировать те взаимодействия, которые будут применяться позже.

Другим полезным видом диаграмм взаимодействий являются коммуникационные диаграммы, которые показывают соединения, и временные диаграммы, показывающие временные интервалы.

CRC–карточки

Одним из наиболее полезных приемов, соответствующих хорошему стилю ООП, является исследование взаимодействия объектов, поскольку его цель состоит в том, чтобы исследовать работу программы, а не данные. CRC-диаграммы (Class-Responsibility-Collaboration, класс-обязанность-кооперация), придуманные Уордом Каннингемом (Ward Cunningham) в конце 80-х годов, выдержали проверку временем и стали высокоэффективным инструментом решения этой задачи (рис. 4.6). И хотя они не входят в состав UML, все же являются очень популярными среди высококвалифицированных разработчиков в области объектных технологий.

Для использования CRC-карточек вы и ваши коллеги должны собраться за столом. Возьмите различные сценарии и проиграйте их с помощью карточек, поднимая их над столом, в то время когда они активны, и передавая их по кругу в предположении, что они посылают сообщение. Эту технологию почти невозможно описать в книге, но легко продемонстрировать; лучший способ научиться этому состоит в том, чтобы попросить кого-нибудь, кто имеет такой опыт, показать вам это.

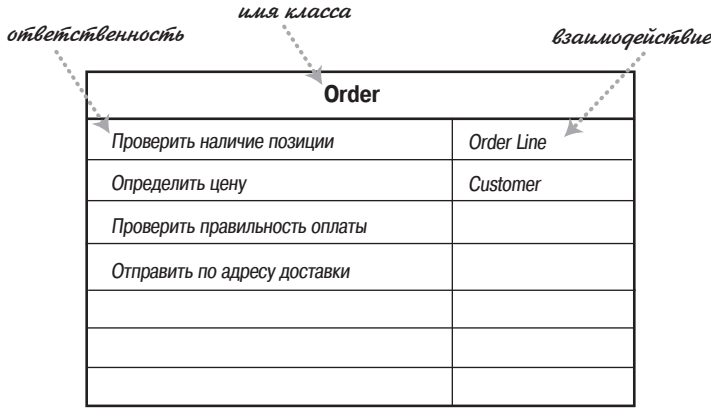

Рис. 4.6. *Пример CRC-карточки*

Важным моментом в CRC-методике является определение ответственностей. **Ответственность** (responsibility) – это краткое описание того, что объект должен делать: операция, которую выполняет объект, некоторый объем знаний, который объект поддерживает, или какие-либо важные решения, принимаемые объектом. Идея состоит в том, чтобы вы могли взять любой класс и сформулировать его разумно ограниченные обязанности. Такой образ действия поможет вам яснее представить себе архитектуру классов.

Вторая буква «С» (в CRC) означает **взаимодействие** (collaboration): другие классы, с которыми должен работать рассматриваемый класс. Это дает вам некоторое представление о связях между классами, но все еще на высоком уровне.

Одно из главных преимуществ CRC-карточек состоит в том, что они способствуют живому обсуждению проектов в среде разработчиков. Если в процессе работы над шаблоном поведения вы хотите посмотреть, как классы его реализуют, то вычерчивание диаграмм взаимодействия, описанных в этой главе, может занять слишком много времени. Обычно требуется просмотреть варианты, а рисование и стирание вариантов на диаграммах может быть очень утомительным. С помощью CRC-карточек разработчики моделируют взаимодействие, поднимая карточки и передавая их по кругу. Это позволяет быстро просчитать варианты.

В процессе работы вы создаете представление об ответственностях и записываете их на карточки. Размышления об ответственностях важны, поскольку рассеивают представление о классах как о бессловесных хранителях данных и помогают членам команды лучше понять поведение каждого класса на высшем уровне. Ответственность может соответствовать операции, атрибуту или, что более точно, неограниченной группе атрибутов и операций.

Мой опыт показывает, что распространенной ошибкой разработчиков является создание ими длинных списков низкоуровневых ответственностей. Это приводит к непониманию сути. Ответственности должны свободно помещаться на одной карточке. Спросите себя, следует ли разделить класс или лучше отрегулировать ответственности, переместив их на более высокий уровень операторов.

Многие разработчики подчеркивают важность ролевой игры, когда каждый член команды играет роль одного или нескольких классов. Я никогда не видел, чтобы Уорд Каннингем так поступал, и думаю, что ролевая игра находится в начале своего пути.

CRC были посвящены целые книги, но не думаю, что они действительно стали сердцем методологии. Первой работой по CRC была статья, написанная Кентом Беком (Kent Beck) [4]. Дополнительную информацию по CRC-карточкам можно найти в [44].

5

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 обозначений для представления различных дополнительных понятий. Я сам обращаюсь к ним не слишком часто, но в отдельных случаях они оказываются весьма удобными. Рассмотрим последовательно эти дополнительные понятия, обращая внимание на особенности их применения.

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

Ключевые слова

Одна из трудностей, сопряженных с графическими языками, состоит в необходимости запоминать значения символов. Когда символов слишком много, пользователям трудно запомнить, что означает каждый из них. Поэтому в UML нередко предпринимаются попытки уменьшить количество символов, заменяя их ключевыми словами. Когда требуется смоделировать конструкцию, отсутствующую в UML, но похожую на один из его элементов, возьмите символ существующей конструкции UML, пометив его ключевым словом, чтобы показать, что используется нечто другое.

Примером может служить интерфейс. **Интерфейс** (interface) в UML (*стр. 96*) означает класс, в котором все операции открытые и не имеют тел методов. Это соответствует интерфейсам в Java, COM (Component Object Module) и CORBA. Поскольку это специальный вид класса, то он изображается с помощью пиктограммы с ключевым словом «inter-

face». Обычно ключевые слова представляются в виде текста, заключенного во французские кавычки («елочки»). Вместо ключевых слов можно использовать специальные значки, но тем самым вы заставляете всех запоминать их значения.

Некоторые ключевые слова, такие как {abstract}, заключаются в фигурные скобки. В действительности никогда не понятно, что формально должно быть в кавычках, а что в фигурных скобках. К счастью, если вы ошибетесь, то заметят это только настоящие знатоки UML. Но лучше быть внимательными.

Некоторые ключевые слова настолько общеупотребительны, что часто заменяются сокращениями: «interface» часто сокращается до «I», а {abstract} – до {A}. Такие сокращения очень полезны, особенно на белых досках, однако их применение не стандартизовано. Поэтому если вы их употребляете, то не забудьте найти место для расшифровки этих обозначений.

В UML 1 кавычки применялись в основном для стереотипов. В UML версии 2 стереотипы определены очень кратко, и разговор о том, что является стереотипом, а что нет, выходит за рамки этой книги. Однако из-за UML 1 многие разработчики употребляют термин «стереотип» в качестве синонима ключевого слова, хотя теперь это неверно.

Стереотипы используются как части профилей. **Профиль** (profile) берет часть UML и расширяет его с помощью связанной группы стереотипов для определенной цели, например для бизнес-моделирования. Полное описание семантики профилей выходит за рамки этой книги. Пока вы не займетесь разработкой серьезной метамодели, вам вряд ли понадобится создавать профиль самому. Скорее всего, вы возьмете профиль, ранее созданный для конкретного варианта моделирования, и, к счастью, применение профиля не требует знания чудовищного количества подробностей, связанных с метамоделью.

Ответственности

Часто бывает удобным показывать ответственности класса (*стр. 90*) на диаграмме классов. Лучший способ показать их состоит в том, чтобы располагать строки комментария в их собственной ячейке (рис. 5.1). При желании ячейке можно присвоить имя, но я обычно этого не делаю, поскольку вероятность возникновения путаницы невелика.

Статические операции и атрибуты

Если в UML ссылаются на операции и атрибуты, принадлежащие классу, а не экземпляру класса, то они называются статическими. Это эквивалентно статическим членам в C-подобных языках. На диаграмме класса статические элементы подчеркиваются (рис. 5.2).

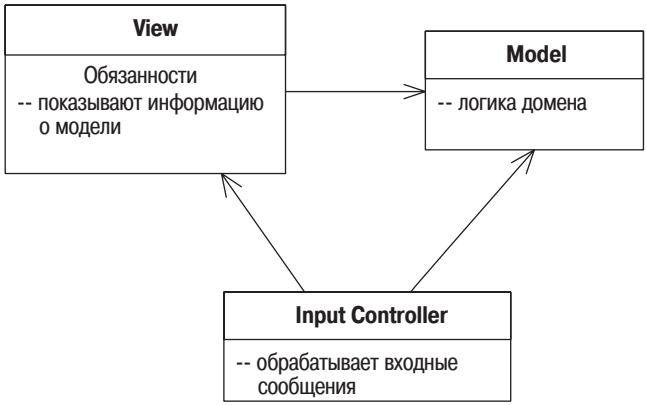

Рис. 5.1. *Представление обязанностей на диаграмме классов*

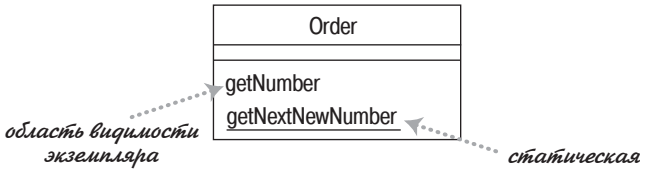

Рис. 5.2. *Статическая нотация*

Агрегация и композиция

К одним из наиболее частых источников недоразумений в UML –можно отнести агрегацию и композицию. В нескольких словах это можно объяснить так: **Агрегация** (aggregation) – это отношение типа «часть целого». Точно так же можно сказать, что двигатель и колеса представляют собой части автомобиля. Звучит вроде бы просто, однако при рассмотрении разницы между агрегацией и композицией возникают определенные трудности.

До появления языка UML вопрос о различии между агрегацией и композицией 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 считают агрегацию важной, но по совершенно другой причине. Язык UML включает агрегацию (рис. 5.3) но семантика ее очень расплывчата. Как говорит Джим Рамбо (Jim Rumbaugh): «Можно представить себе агрегацию как плацебо для моделирования» [40].

Наряду с агрегацией в языке UML есть более определенное свойство – **композиция** (composition). На рис. 5.4 экземпляр класса `Point` (Точка) может быть частью многоугольника, а может представлять центр окружности, но он не может быть и тем и другим одновременно. Главное

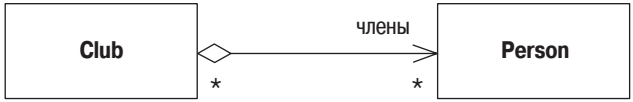

Рис. 5.3. *Агрегация*

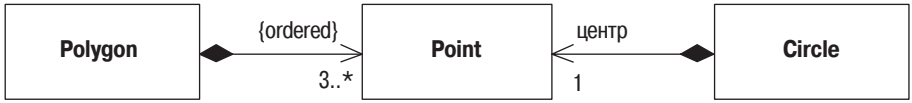

Рис. 5.4. *Композиция*

правило состоит в том, что хотя класс может быть частью нескольких других классов, но любой экземпляр может принадлежать только одному владельцу. На диаграмме классов можно показать несколько классов потенциальных владельцев, но у любого экземпляра класса есть только один объект-владелец.

Вы заметите, что на рис 5.4 я не показываю обратные кратности. В большинстве случаев, как и здесь, они равны 0..1. Единственной альтернативой является значение 1, когда класс-компонент разработан таким образом, что у него только один класс-владелец.

Правило «нет совместного владения» является ключевым в композиции. Другое допущение состоит в том, что если удаляется многоугольник (Polygon), то автоматически должны удалиться все точки (Points), которыми он владеет.

Композиция – это хороший способ показать свойства, которыми владеют по значению, свойства объектов-значений (*стр. 100*) или свойства, которые имеют определенные и до некоторой степени исключительные права владения другими компонентами. Агрегация совершенно не имеет смысла; поэтому я не рекомендовал бы применять ее в диаграммах. Если вы встретите ее в диаграммах других разработчиков, то вам придется покопаться, чтобы понять их значение. Разные авторы и команды разработчиков используют их в совершенно разных целях.

Производные свойства

Производные свойства (derived properties) могут вычисляться на основе других значений. Говоря об интервале дат (рис. 5.5), мы можем рассуждать о трех свойствах: начальной дате, конечной дате и количестве дней за данный период. Эти значения связаны, поэтому мы можем сказать, что длина является производной двух других значений.

С точки зрения программного обеспечения образование производных можно интерпретировать двумя различными путями. Можно использовать образование производных для обозначения различия между вычисляемым и хранимым значениями. В этом случае, глядя на рис. 5.5,

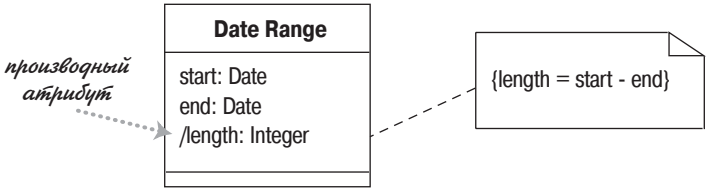

Рис. 5.5. *Производный атрибут для временного интервала*

мы скажем, что начальная (start) и конечная (end) даты хранятся, а длина (length) вычисляется. И хотя это наиболее распространенное применение, меня это не очень привлекает, поскольку слишком раскрывает внутреннее устройство класса DateRange (Интервал дат).

Я предпочитаю рассматривать это как связь между значениями. В данном случае мы говорим, что между тремя значениями существует связь, но не важно, какое из трех значений вычисляется. При этом можно произвольно выбирать, какой атрибут отмечать как производный, а можно и вовсе этого не делать, но все же полезно напомнить разработчикам о связи. Такое применение имеет смысл в концептуальных диаграммах.

Образование производных может быть применено к свойствам с помощью ассоциаций. В этом случае вы просто отмечаете имя символом «/».

Интерфейсы и абстрактные классы

Абстрактный класс (abstract class) – это класс, который нельзя реализовать непосредственно. Вместо этого создается экземпляр подкласса. Обычно абстрактный класс имеет одну или более абстрактных операций. У **абстрактной операции** (abstract operation) нет реализации; это чистое объявление, которое клиенты могут привязать к абстрактному классу.

Наиболее распространенным способом обозначения абстрактного класса или операции в языке UML является написание их имен курсивом. Можно также сделать свойства абстрактными, определяя абстрактное свойство или методы доступа. Курсив сложно изобразить на доске, поэтому можно прибегнуть к метке: {abstract}.

Интерфейс – это класс, не имеющий реализации, то есть вся его функциональность абстрактна. Интерфейсы прямо соответствуют интерфейсам в C# и Java и являются общей идиомой в других типизированных языках. Интерфейс обозначается ключевым словом «interface».

Классы обладают двумя типами отношений с интерфейсами: предоставление или требование. Класс **предоставляет интерфейс**, если его можно заменить на интерфейс. В Java и .NET класс может сделать это, реализуя интерфейс или подтип интерфейса. В C++ создается подкласс класса, являющегося интерфейсом.

Класс **требует интерфейс**, если для работы ему нужен экземпляр данного интерфейса. По сути дела, это зависимость от интерфейса.

На рис. 5.6 эти отношения демонстрируются в действии на базе небольшого набора классов, заимствованных из Java. Я мог бы написать класс `Order` (Заказ), содержащий список позиций заказа (`Line Items`). Поскольку я использую список, то класс `Order` зависит от интерфейса `List` (Список). Предположим, что он вызывает методы `equals`, `add` и `get`. При выполнении связывания объект `Order` действительно будет использовать экземпляр класса `ArrayList`, но ему не нужно знать, что необходимо вызывать эти три метода, поскольку они входят в состав интерфейса `List`.

Класс `ArrayList` – это подкласс класса `AbstractList`. Класс `AbstractList` предоставляет некоторую, но не всю реализацию поведения интерфейса `List`. В частности, метод `get` – абстрактный. В результате `ArrayList` реализует метод `get`, а также переопределяет некоторые другие операции класса `AbstractList`. В данном случае он переопределяет метод `add`, но вполне удовлетворен наследованием реализации метода `equals`.

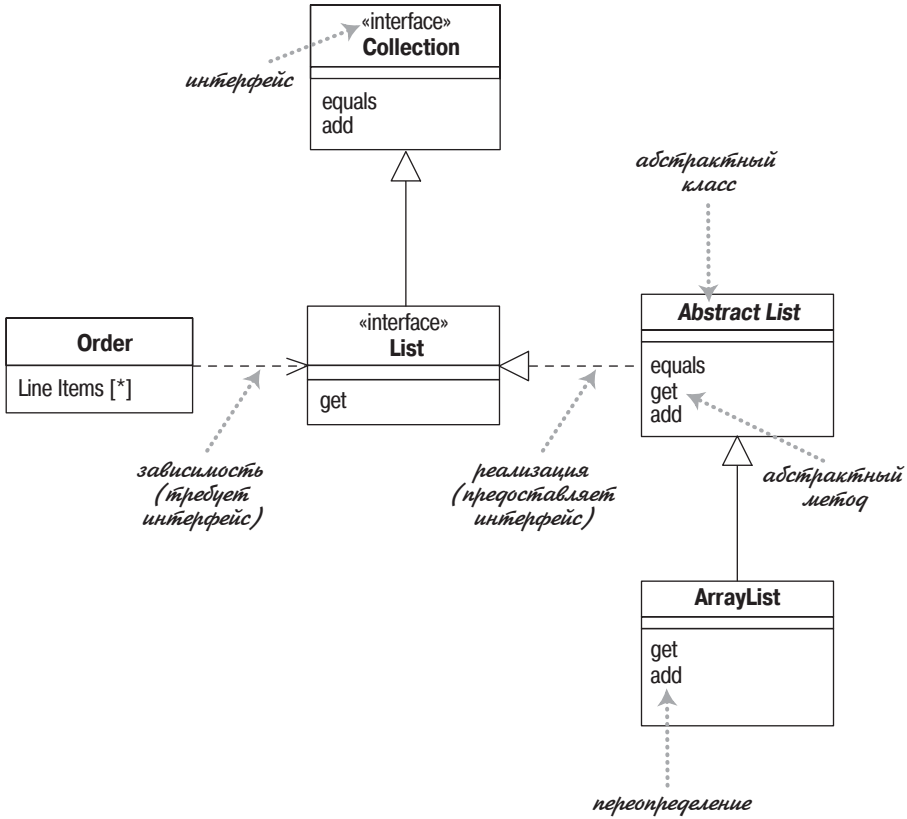

Рис. 5.6. *Пример интерфейсов и абстрактного класса на языке Java*

Почему бы мне просто не отказаться от этого и не заставить Order прямо использовать ArrayList? Применение интерфейса позволяет мне получить преимущество при последующем изменении реализации, если потребуется. Другой способ реализации может оказаться более производительным – он может предоставить функции работы с базой данных или другие возможности. Программируя интерфейс, а не реализацию, я избегаю необходимости переделывать весь код, когда достаточно изменить реализацию класса List. Следует всегда стараться программировать интерфейс так, как показано выше, то есть всегда использовать наиболее общий тип.

Относительно вышесказанного приведу один практический совет. Когда программисты применяют коллекцию, подобную приведенной здесь, они обычно инициализируют ее при объявлении, например:

```
private List lineItems = new ArrayList();
```

Обратите внимание, что это определенно приводит к зависимости Order от конкретного ArrayList. С точки зрения теории это проблема, но на практике разработчиков это не беспокоит. Поскольку lineItems объявлен как List, то никакая другая часть класса Order не зависит от ArrayList. При необходимости изменить реализацию нужно побеспокоиться лишь об одной строке кода инициализации. Общепринято ссылаться на конкретный класс единожды – при создании, а впоследствии использовать только интерфейс.

Полная нотация на рис. 5.6 – это один из способов обозначения интерфейса. На рис. 5.7 показана более компактная нотация. Тот факт, что ArrayList реализует List и Collection, показан с помощью кружков, называемых часто «леденцами на палочках». То, что Order требует интерфейс List, показано с помощью значка «гнездо». Связь совершенно очевидна.

В UML уже применялась нотация «леденцов на палочках», но гнездовая нотация – это новинка UML 2. (Мне кажется, это моя любимая нотация из добавленных.) Возможно, вы встретите более старые диаграммы, использующие стиль, представленный на рис. 5.8, где зависимость основана на нотации леденцов.

Любой класс – это сочетание интерфейса и реализации. Поэтому мы часто можем видеть, что объект используется посредством интерфейса одного из его суперклассов. Определенно, было бы допустимо исполь-

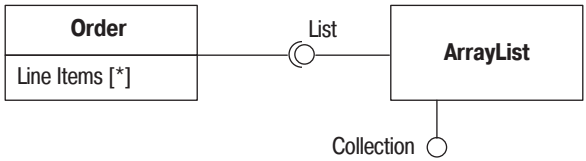

Рис. 5.7. *Шарово-гнездовая нотация*

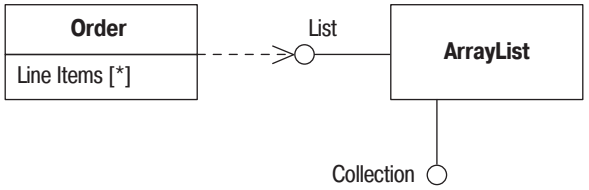

Рис. 5.8. *Более старое обозначение зависимостей с помощью «леденцов на палочке»*

зовать для суперкласса нотацию леденцов, поскольку суперкласс – это класс, а не чистый интерфейс. Но я обхожу эти правила для ясности.

Разработчики сочли, что нотация леденцов полезна не только для диаграмм классов, но и в других местах. Одна из вечных проблем диаграмм взаимодействий заключается в том, что они не обеспечивают хорошую визуализацию полиморфного поведения. Хотя это нормативное применение, вы можете обозначить такое поведение вдоль линий, как на рис. 5.9. Здесь, как вы можете видеть, хотя у нас есть экземпляр класса `Salesman`, который используется объектом `Bonus Calculator` как таковой, но объект `Pay Period` использует `Salesman` только через его интерфейс `Employee`. (Тот же самый прием может применяться и в случае коммуникационных диаграмм.)

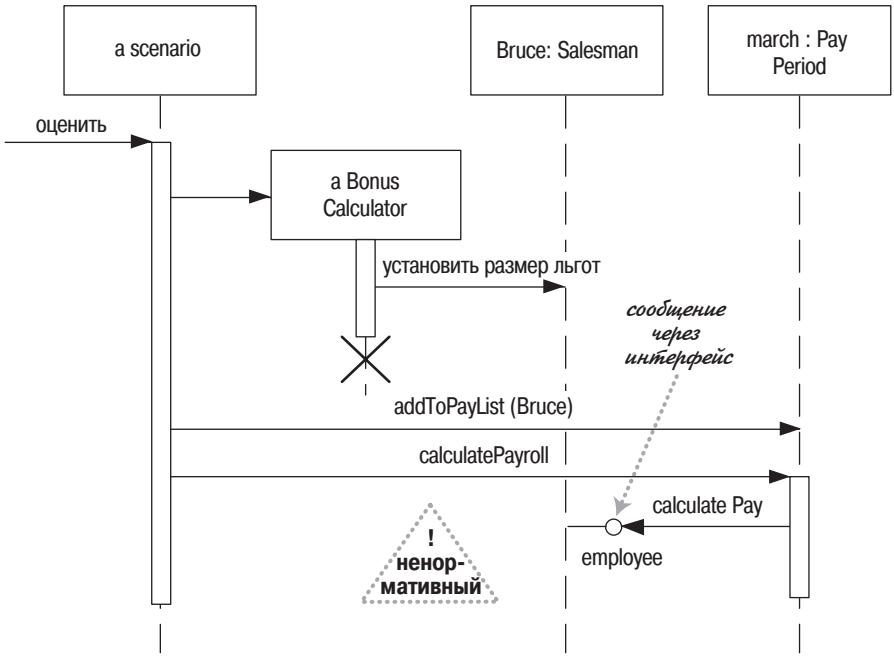

Рис. 5.9. *Представление полиморфизма на диаграммах последовательности с помощью нотации леденцов*

Read-Only и Frozen

На *стр. 64* я описал ключевое слово {readOnly} (только для чтения). Этим ключевым словом обозначается свойство, которое клиенты могут только читать, но не могут обновлять. Подобное, но несколько отличающееся ключевое слово {frozen} (замороженный) было в UML 1. Свойство находится в состоянии frozen, если оно не может быть изменено в течение жизни объекта; такие свойства часто называются неизменными (immutable). Оно было исключено из UML 2, но понятие frozen очень полезное, поэтому я буду применять его по-прежнему. Наряду с обозначением отдельных свойств, ключевое слово frozen можно применять к классу для указания того, что все свойства всех экземпляров класса находятся в состоянии frozen. (До меня дошли слухи, что ключевое слово frozen скоро будет восстановлено.)

Объекты-ссылки и объекты-значения

Одна из наиболее общих черт объектов заключается в том, что они обладают индивидуальностью (identity). Это правда, но все обстоит не столь просто, как может показаться. На практике оказывается, что индивидуальность важна для объектов-ссылок, но она не так важна для объектов-значений.

Объекты-ссылки (reference objects) – это такие объекты, как Customer (Клиент). В данном случае индивидуальность очень важна, поскольку в реальном мире конкретному классу обычно должен соответствовать только один программный объект. Любой объект, который обращается к объекту Customer, может воспользоваться соответствующей ссылкой или указателем. В результате все объекты, обращающиеся к данному объекту Customer, получат доступ к одному и тому же программному объекту. Таким образом, изменения, вносимые в объект Customer, будут доступны всем пользователям данного объекта.

Если имеются две ссылки на объект Customer и требуется установить их тождественность, то обычно сравниваются индивидуальности тех объектов, на которые указывают эти ссылки. Создание копий может быть запрещено; если же оно разрешено, то, как правило, применяется редко – возможно, для архивирования или репликации через компьютерную сеть. Если копии созданы, то необходимо обеспечить синхронизацию вносимых в них изменений.

Объекты-значения (value objects) – это такие объекты, как Date (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 2004 года». Все эти объекты являются взаимозаменяемыми копиями. При этом новые даты создаются и уничтожаются достаточно часто.

Если имеются две даты и надо установить, тождественны ли они, 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 в датах год, месяц и день (каким бы ни было их внутреннее представление). Обычно каждый объект, ссылающийся на 1 января 2004 года, имеет собственный специальный объект, однако иногда даты могут быть объектами общего пользования.

Объекты-значения должны быть постоянными. Другими словами, не должно допускаться изменение значения объекта-даты «1 января 2004 года» на «2 января 2004 года». Вместо этого следует создать новый объект «2 января 2004 года» и использовать его вместо первого объекта. Причина запрета подобного изменения заключается в следующем: если бы эта дата была объектом общего пользования, то ее обновление могло бы повлиять на другие объекты непредсказуемым образом. Данная проблема известна как **совмещение имен** (aliasing).

В прежнее время различие между объектами-ссылками и объектами-значениями было более четким. Объекты-значения являлись встроенными элементами системы типов. В настоящее время можно расширить систему типов с помощью собственных классов, поэтому данный аспект требует более внимательного отношения.

В языке UML используется концепция **типа данных**, который представляется ключевым словом на символе класса. Строго говоря, тип данных не идентичен объекту-значению, поскольку типы данных не могут иметь индивидуальности. Объекты-значения могут иметь индивидуальность, но она не используется для проверки равенства. В языке Java примитивы могут быть типами данных, но не даты, хотя они могут быть объектами-значениями. Если требуется их выделить, то при создании взаимосвязи с объектом-значением я использую композицию. Можно также применить ключевое слово для типа значения; на мой взгляд, стандартными являются слова «`value`» и «`struct`».

Квалифицированные ассоциации

Квалифицированная ассоциация в языке UML эквивалентна таким известным понятиям в языках программирования, как ассоциативные массивы (associative arrays), проекции (maps), хеши (hashes) и словари (dictionaries). Рисунок 5.10 иллюстрирует способ представления ассоциации между классами `Order` (Заказ) и `Order Line` (Строка заказа), в котором используется квалификатор. Квалификатор указывает, что в соответствии с заказом для каждого экземпляра продукта (`Product`) может существовать только одна строка заказа.

С точки зрения программного обеспечения такая квалифицированная ассоциация может повлечь создание интерфейса следующего вида:

***Рис. 5.10.** Квалифицированная ассоциация*

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

Таким образом, любой доступ к определенной строке заказа требует подстановки некоторого продукта в качестве аргумента, в предположении, что это структура данных, состоящая из ключа и значения.

Разработчиков часто ставит в тупик кратность квалифицированных ассоциаций. На рис. 5.10 заказ может иметь несколько позиций заказа (`Line Items`), но кратность квалифицированной ассоциации – это кратность в контексте квалификатора. Поэтому диаграмма говорит, что в заказе имеется 0..1 позиций заказа на продукт. Кратность, равная 1, означает, что в заказе должна быть одна позиция заказа для каждого продукта. Кратность * означает, что для любого продукта может существовать несколько позиций заказа, но доступ к позициям заказа индексируется по продукту.

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».

Классификация и обобщение

Мне часто приходится слышать суждения разработчиков о механизме подтипов как об отношении *является* (*это* [*есть*]). Я настоятельно рекомендую держаться подальше от такого представления. Проблема заключается в том, что выражение *является* может иметь самый разный смысл.

Рассмотрим следующие предложения.

1. Шеп – это бордер-колли.
2. Бордер-колли – это собака.
3. Собаки являются животными.
4. Бордер-колли – это порода собак.
5. Собака – это биологический вид.

Теперь попытаемся скомбинировать эти фразы. При объединении первого и второго предложений получаем «Шеп – это собака»; второе и третье предложения в результате дают «бордер-колли – это живот-

ные». Объединение первых трех фраз дает «Шеп – это животное». Чем дальше, тем лучше. Теперь попробуем первое и четвертое предложения: «Шеп – это порода собак». В результате объединения второго и пятого предложений получим «бордер-колли – это биологический вид». Это уже не так хорошо.

Почему некоторые из этих фраз можно комбинировать, а другие нельзя? Причина в том, что некоторые предложения представляют собой **классификацию** – объект Шеп (Shep) является экземпляром типа Бордер-Колли (Border Collie), в то время как другие предложения представляют собой **обобщение** – тип Бордер-Колли является подтипом типа Собака (Dog). Обобщение транзитивно, а классификация – нет. Если обобщение следует за классификацией, то их можно объединить, а если наоборот – классификация следует за обобщением, то нельзя.

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

В языке UML обобщение обозначается соответствующим символом. Для того чтобы показать классификацию, применяется зависимость с ключевым словом «instantiate».

Множественная и динамическая классификация

Классификация служит для обозначения отношения между некоторым объектом и его типом. В основных языках программирования предполагается, что объект относится к единственному классу. Но в UML имеется больше возможностей для классификации.

При **однозначной классификации** (single classification) любой объект принадлежит единственному типу, который может быть унаследован от супертипов. Во **множественной классификации** (multiple classification) объект может быть описан несколькими типами, которые не обязательно должны быть связаны наследованием.

Множественная классификация отличается от множественного наследования. При множественном наследовании тип может иметь несколько супертипов, но для каждого объекта должен быть только один тип. Множественная классификация допускает принадлежность объекта нескольким типам, при этом не требуется определять специальный тип.

В качестве примера рассмотрим тип `Person` (Личность), подтипами которого являются `Male` (Мужчина) или `Female` (Женщина), `Doctor` (Доктор) или `Nurse` (Медсестра), `Patient` (Пациент) или вообще никто (рис. 5.11). Множественная классификация позволяет некоторому

объекту иметь любой из этих типов в любом допустимом сочетании, при этом нет необходимости определять отдельные типы для всех возможных комбинаций.

Если вы используете множественную классификацию, то должны быть уверены в том, что четко определили, какие комбинации являются допустимыми. В языке UML версии 2 это осуществляется помещением каждого обобщающего отношения в **множество обобщения**. На диаграмме классов вы помечаете линию обобщения с помощью имени множества обобщения, которое в UML 1 называется дискриминатором. Единственная классификация соответствует одному безымянному множеству обобщения.

По определению множества обобщения не пересекаются: каждый экземпляр супертипа может быть экземпляром только одного подтипа из данного множества. Если вы соединяете линии обобщений с одной стрелкой, то они должны входить в одно и то же множество обобщения, как показано на рис. 5.11. Альтернативный способ – изобразить несколько стрелок с одинаковой текстовой меткой.

В качестве иллюстрации отметим на диаграмме следующие допустимые комбинации подтипов: (`Female`, `Patient`, `Nurse`); (`Male`, `Physiotherapist` (Физиотерапевт)); (`Female`, `Patient`) и (`Female`, `Doctor`, `Surgeon` (Хирург)). Комбинация (`Patient`, `Doctor`, `Nurse`) является недопустимой, поскольку она содержит два типа из множества обобщения `role` (роль).

Возникает еще один вопрос: может ли объект изменить свой класс? Например, когда банковский счет клиента становится пустым, он существенно меняет свое поведение. В частности, отклоняются некоторые операции, такие как «снять со счета» и «закрыть счет».

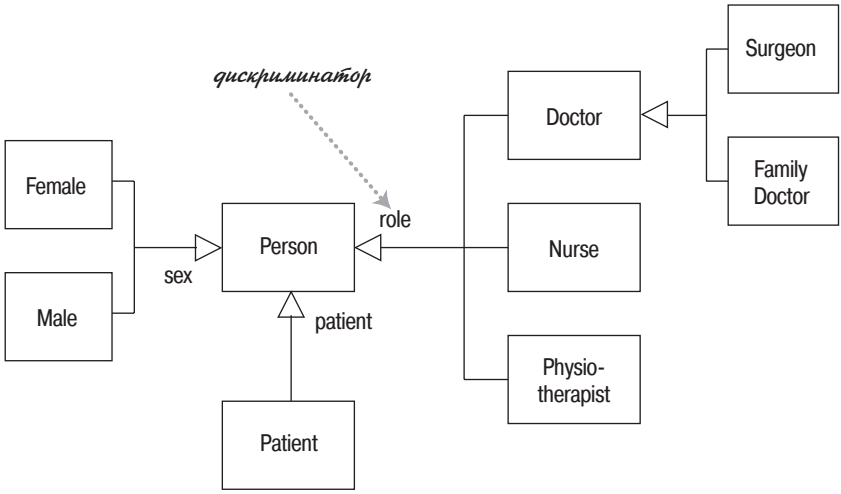

Рис. 5.11. *Множественная классификация*

Динамическая классификация (dynamic classification) разрешает объектам изменять свой тип в рамках структуры подтипов, а **статическая классификация** (static classification) 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. Однако с точки зрения программного обеспечения на пути ее реализации слишком много препятствий. В подавляющем большинстве диаграмм UML вы встретите только однозначную статическую классификацию, поэтому она должна стать вашим обычным инструментом.

Класс-ассоциация

Классы-ассоциации (association classes) позволяют дополнительно определять для ассоциаций атрибуты, операции и другие свойства, как показано на рис. 5.12. Из данной диаграммы видно, что `Person` (Личность) может принимать участие в нескольких совещаниях (`Meeting`). При этом необходимо каким-то образом хранить информацию о том, насколько внимательной была данная личность; это можно сделать, добавив к ассоциации атрибут `attentiveness` (внимательность).

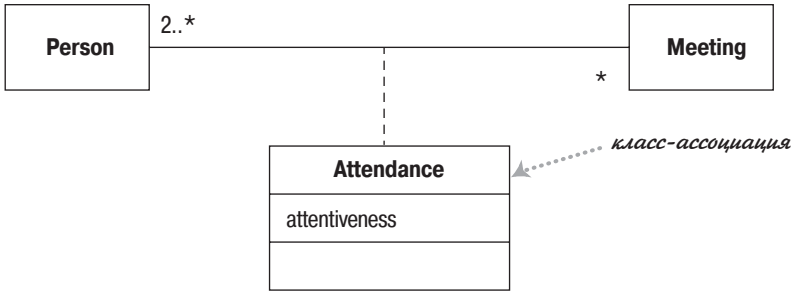

Рис. 5.12. *Класс-ассоциация*

На рис. 5.13 показан другой способ представления данной информации: образование самостоятельного класса `Attendance` (Присутствие). Обратите внимание, как при этом изменили свои значения кратности.

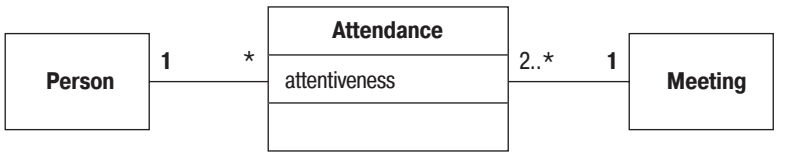

Рис. 5.13. *Развитие класса-ассоциации до обычного класса*

Какие же преимущества 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, необходимо привести еще один пример.

Посмотрим на две диаграммы, изображенные на рис. 5.14. Форма этих диаграмм практически одинакова. Хотя можно себе представить компанию (`Company`), играющую различные роли (`Role`) в одном и том же контракте (`Contract`), но трудно вообразить личность (`Person`), имеющую различные уровни компетенции в одном и том же навыке (`Skill`); действительно, скорее всего, это можно считать ошибкой.

В языке UML допустим только последний вариант. Может существовать только один уровень компетенции для каждой комбинации личности и навыка. Верхняя диаграмма на рис. 5.14 не допускает участия компании более чем в одной роли в одном и том же контракте. Если без этого не обойтись, надо превратить `Role` в полный класс, как это сделано на рис. 5.13.

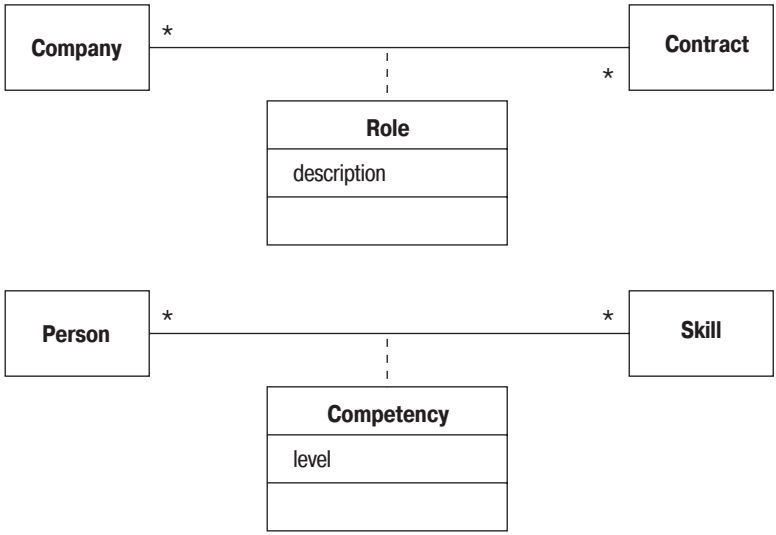

Рис. 5.14. *Хитрости класса-ассоциации (класс Role, возможно, не должен быть классом-ассоциацией)*

Реализация классов-ассоциаций не слишком очевидна. Мой совет: реализовывать класс-ассоциацию так, как будто это обычный класс, но методы, предоставляющие информацию связанным классам, должны принадлежать классу-ассоциации. Поэтому для случая, изображенного на рис. 5.12, я бы представил следующие методы класса `Person`:

```
class Person
List getAttendances()
List getMeetings()
```

Таким образом, клиенты объекта Person могут обнаружить сотрудников на совещании; если им требуются детали, то они могут получить собственно часы работы (Attendance). Если вы так делаете, не забудьте об ограничении, при котором для любой пары объектов Person (Личность) и Meeting (Совещание) может существовать только один объект Attendance (Присутствие).

Часто этот вид конструкции можно встретить там, где речь идет о временных изменениях (см., например, рис. 5.15). Однако я считаю, что создание дополнительных классов или классов-ассоциаций может сделать модель сложной для понимания, а также направить реализацию в неправильное русло.

Рис. 5.15. *Использование класса для временного отношения*

Если я встречаю временную информацию такого типа, то использую для ассоциации ключевое слово «temporal» (временной) (рис. 5.16). Модель означает, что некоторое время личность может работать только в одной компании. Однако по прошествии времени личность сможет работать в нескольких компаниях. Это предполагает интерфейс, описываемый следующими строками:

```
class Person ...
Company getEmployer();         // определение текущего работодателя
Company getEmployer(Date);     // определение работодателя на указанный момент
void changeEmployer(Company newEmployer,Date changeDate);
void leaveEmployer (Date changeDate);
```

Ключевое слово «temporal» не входит в состав языка UML, но я упомянул его здесь по двум причинам. Во-первых, это понятие часто оказывалось полезным для меня как проектировщика. Во-вторых, это демонстрирует, как можно применять ключевые слова для расширения языка UML. Дополнительную информацию по данному вопросу можно найти на *http://martinfowler.com/ap2/timeNarrative.html*.

Рис. 5.16. *Ключевое слово «temporal» для ассоциаций*

Шаблон класса (параметризованный класс)

Некоторые языки, в особенности C++, включают в себя понятие **параметризованного класса** (parameterized class) или **шаблона** (template). (Шаблоны могут быть включены в языки Java и C# в ближайшем будущем.)

Наиболее очевидная польза от применения этого понятия проявляется при работе с коллекциями в строго типизированных языках. Таким образом, в общем случае поведение для множества можно определить с помощью шаблона класса `Set` (Множество).

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

После этого можно использовать данное общее определение для задания более конкретных классов-множеств:

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

Для объявления класса-шаблона в языке UML используется нотация, показанная на рис. 5.17. Прямоугольник с буквой «T» на диаграмме служит для указания параметра типа. (Можно указать более одного параметра.)

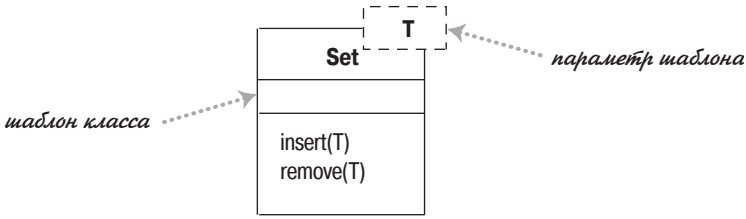

Рис. 5.17. *Класс-шаблон*

Применение параметризованного класса, такого как `Set<Employee>`, называется **образованием производных** (derivation). Образование производных можно изобразить двумя способами. Первый способ отражает синтаксис языка C++ (рис. 5.18). Выражение образования производных описывается в угловых скобках в виде `<parameter-name::parameter-value>`. Если указывается только один параметр, то обычно имя параметра опускают. Альтернативная нотация (рис. 5.19) усиливает связь с шаблоном и допускает переименование связанного элемента.

Рис. 5.18. *Связанный элемент (вариант 1)*

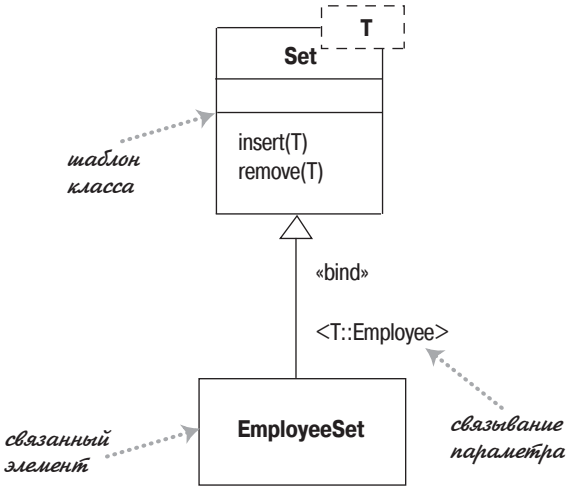

Рис. 5.19. *Связанный элемент (вариант 2)*

Ключевое слово «bind» (связать) является стереотипом отношения уточнения. Это отношение означает, что класс EmployeeSet (Множество служащих) будет согласован с интерфейсом класса Set. Можете считать EmployeeSet подтипом типа Set. Это соответствует другому способу реализации типизированных коллекций, который заключается в объявлении всех соответствующих подтипов.

Однако использование образования производных не эквивалентно определению подтипа. В связанный элемент нельзя добавлять новые возможности – он полностью определен своим шаблоном; можно только добавить информацию, ограничивающую его тип. Если же вы хотите добавить возможности, то должны создать некоторый подтип.

Перечисления

Перечисления (рис. 5.20) используются для представления фиксированного набора значений, у которых нет других свойств кроме их символических значений. Они изображаются в виде класса с ключевым словом «enumeration».

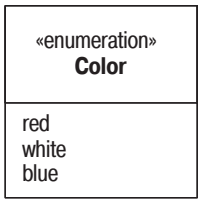

Рис. 5.20. *Перечисление*

Активный класс

Активный класс (active class) имеет экземпляры, каждый из которых выполняет и управляет собственным потоком управления. Вызовы методов могут выполняться в клиентском потоке или в потоке активного объекта. Удачным примером может служить командный процессор, который принимает извне командные объекты, а затем исполняет команды в контексте собственного потока управления.

Как видно из рис. 5.21, при переходе от UML 1 к UML 2 нотация активных классов изменилась. В UML 2 активный класс обозначен дополнительными вертикальными линиями по краям; в UML 1 он имел толстую границу и назывался активным объектом.

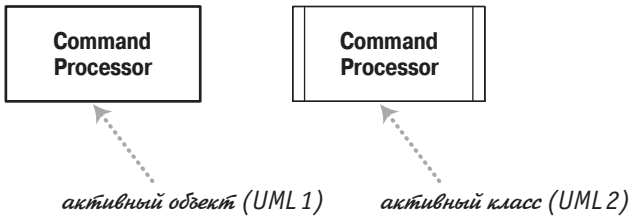

***Рис. 5.21.** Активный класс*

Видимость

Видимость (visibility) – это понятие простое по существу, но содержащее сложные нюансы. Идея заключается в том, что у любого класса имеются открытые (public) и закрытые (private) элементы. Открытые элементы могут быть использованы любым другим классом, а закрытые элементы – только классом-владельцем. Однако в каждом языке действуют свои правила. Несмотря на то что во многих языках употребляются такие термины, как *public* (открытый), *private* (закрытый) и *protected* (защищенный), в разных языках они имеют различные значения. Эти различия невелики, но они приводят к недоразумениям, особенно у тех, кто использует в своей работе более одного языка программирования.

В языке UML делается попытка решить эту задачу, не устраивая жуткую путаницу. По существу, в рамках UML для любого атрибута или операции можно указать индикатор видимости. Для этой цели можно использовать любую подходящую метку, смысл которой определяется тем или иным языком программирования. Тем не менее в UML предлагается четыре аббревиатуры для обозначения видимости: + (public – открытый), - (private – закрытый), ~ (package – пакетный) и # (protected – защищенный). Эти четыре уровня определены и используются в рамках метамодели языка UML, но их определения очень незначительно отличаются от соответствующих определений в других языках программирования.

При использовании видимости применяйте правила того языка программирования, на котором вы работаете. Рассматривая модель в UML с какой-нибудь точки зрения, аккуратно расшифровывайте значения маркеров видимости и старайтесь понять, как эти значения могут измениться при переходе от одного языка программирования к другому.

В подавляющем большинстве случаев я не рисую на диаграммах маркеры видимости; я их использую, только если необходимо подчеркнуть различия в видимости определенных свойств. И даже в таких случаях я могу, как правило, обойтись без + и -, которые, по крайней мере, достаточно легко запомнить.

Сообщения

В стандартном языке UML не отображается информация о сообщениях на диаграммах классов. Однако иногда я встречал диаграммы, подобные той, которая приведена на рис. 5.22.

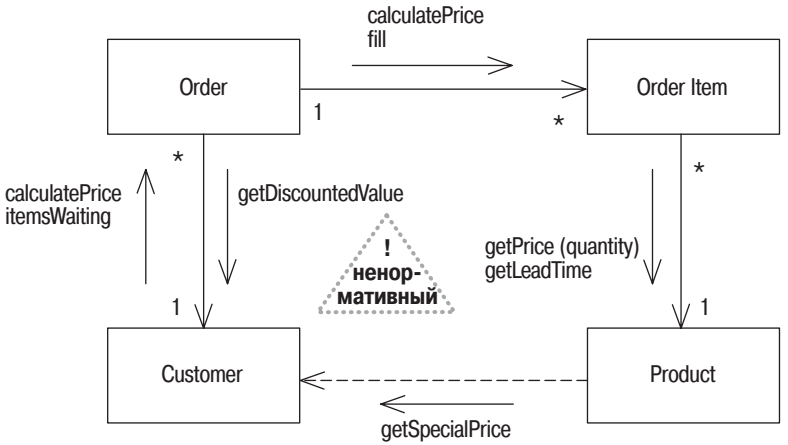

Рис. 5.22. *Классы с сообщениями*

При этом добавляются стрелки со стороны ассоциаций. Стрелки помечаются сообщениями, которые один объект посылает другому. Поскольку для посылки сообщения классу наличие ассоциации с ним не обязательно, то может потребоваться дополнительная стрелка зависимости, чтобы отобразить сообщения между классами, не связанными ассоциацией.

Информация о сообщениях охватывает несколько прецедентов, поэтому, в отличие от коммуникационных диаграмм, они не нумеруются, чтобы показать их последовательность.

6

Диаграммы объектов

Диаграмма объектов (object diagram) – это снимок объектов системы в какой-то момент времени. Поскольку она показывает экземпляры, а не классы, то диаграмму объектов часто называют диаграммой экземпляров.

Диаграмму объектов можно использовать для отображения одного из вариантов конфигурации объектов. (На рис. 6.1 показано множество классов, а на рис. 6.2 представлено множество связанных объектов.) Последний вариант очень полезен, когда допустимые связи между объектами могут быть сложными.

Можно определить, что элементы, показанные на рис. 6.2, являются экземплярами, поскольку их имена подчеркнуты. Каждое имя представляется в виде: `имя экземпляра : имя класса`. Обе части имени не являются обязательными, поэтому имена `John`, `:Person` и `aPerson` являются допустимыми. Если указано только имя класса, то необходимо поставить двоеточие. Можно также задать значения и атрибуты, как показано на рис. 6.2.

Строго говоря, элементы диаграммы объектов – это спецификации экземпляров, а не сами экземпляры. Причина в том, что разрешается оставлять обязательные атрибуты пустыми или показывать спецификации экземпляров абстрактных классов. Можно рассматривать **спе-**

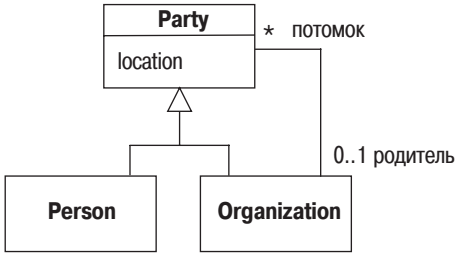

Рис. 6.1. *Диаграмма классов, показывающая структуру класса Party (вечеринка)*

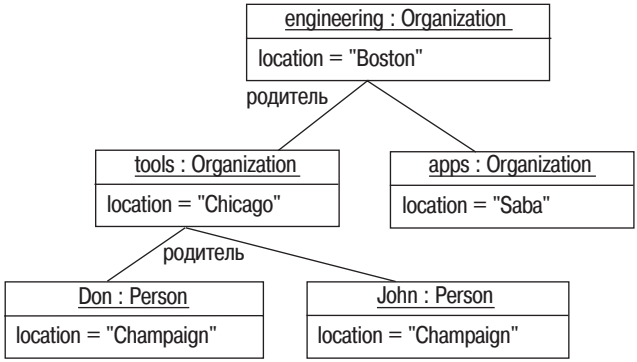

Рис. 6.2. *Диаграмма объектов с примером экземпляра класса Party*

цификации экземпляров (instance specifications**)** как частично определенные экземпляры.

С другой стороны, диаграмму объектов можно считать коммуникационной диаграммой **(***стр. 152***)** без сообщений.

Когда применяются диаграммы объектов

Диаграммы объектов удобны для показа примеров связанных друг с другом объектов. Во многих ситуациях точную структуру можно определить с помощью диаграммы классов, но при этом структура остается трудной для понимания. В таких случаях пара примеров диаграммы объектов может прояснить ситуацию.

7

Диаграммы пакетов

Классы составляют структурный костяк объектно-ориентированной системы. Хотя они исключительно полезны, но нужно нечто большее для структурирования больших систем, которые могут состоять из сотен классов.

Пакет (package) – это инструмент группирования, который позволяет взять любую конструкцию UML и объединить ее элементы в единицы высокого уровня. В основном пакеты служат для объединения классов в группы, и именно этот способ их применения я здесь описываю, но помните, что пакеты могут применяться для любой другой конструкции языка UML.

В модели UML каждый класс может включаться только в один пакет. Пакеты могут также входить в состав других пакетов, поэтому мы остаемся в иерархической структуре, в которой пакеты верхнего уровня распадаются на подпакеты со своими собственными подпакетами, и так далее, до самого низа иерархии классов. Пакет может содержать и подпакеты, и классы.

В терминах программирования пакеты в UML соответствуют таким группирующим конструкциям, как пакеты в Java и пространства имен в C++ и .NET.

Каждый пакет представляет **пространство имен** (namespace), а это означает, что каждый класс внутри собственного пакета должен иметь уникальное имя. Если я хочу создать пакет с именем `Date`, а класс `Date` уже существует в пакете `System`, то я обязан поместить его в отдельный пакет. Чтобы отличить один класс от другого, я могу использовать **полностью определенное имя** (fully qualified name), то есть имя, которое указывает на структуру, владеющую пакетом. В языке UML в именах пакетов используются двойные двоеточия, поэтому классы дат могут иметь имена `System::Date` и `MartinFowler::Util::Date`.

На диаграммах пакеты изображаются в виде папок с закладками (рис. 7.1). Можно показывать только имя пакета или имя вместе с его

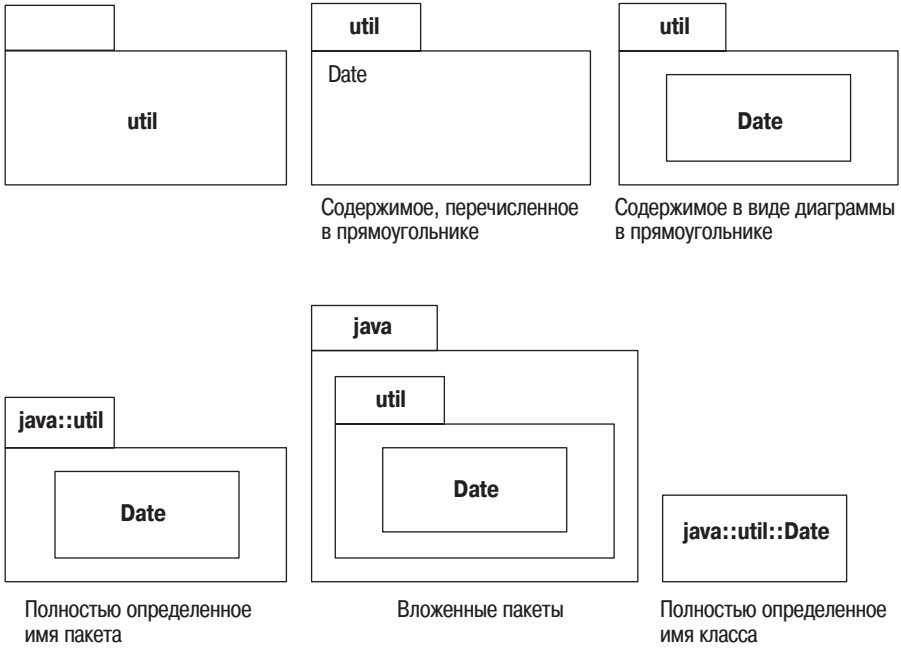

Рис. 7.1. *Способы изображения пакетов на диаграммах*

содержимым. В любом случае можно использовать либо полностью определенные имена, либо обычные имена. Изображение содержимого с помощью значков классов позволяет показать все особенности класса, вплоть до изображения диаграммы классов внутри пакета. Простое перечисление имен имеет смысл, если вы хотите лишь показать, какие классы входят в той или иной пакет.

Вполне можно встретить класс, например с именем `Date` (как в java.util), а не с полностью определенным именем. Этот стиль является соглашением, в основном принятым в Rational Rose, которое не входит в стандарт языка.

UML разрешает классам в пакетах быть открытыми (public) или закрытыми (private). Открытые классы представляют часть интерфейса пакета и могут быть использованы классами из других пакетов; закрытые классы недоступны извне. В различных средах программирования действуют различные правила в отношении видимости их группирующими конструкциями; необходимо придерживаться правил своего программного окружения, даже если это идет вразрез с правилами UML.

В таких случаях полезно сократить интерфейс пакета, экспортируя только небольшое подмножество операций, связанных с открытыми классами пакета. Можно сделать это, присвоив всем классам модификатор видимости private (закрытый), так чтобы они были доступны

только классам данного пакета, а также создав дополнительные открытые классы для внешнего использования. Эти дополнительные классы, называемые *Facades* (Фасады) [21], делегируют открытые операции своим более застенчивым соратникам по пакету.

Как распределить классы по пакетам? Это действительно сложный вопрос, на который может ответить только специалист с большим опытом работы в области проектирования. В этом деле могут помочь два правила: общий принцип замыкания (Common Closure Principle) и общий принцип повторного использования (Common Reuse Principle) [30]. Общий принцип замыкания гласит, что причины изменения классов пакета должны быть одинаковые. Общий принцип повторного использования утверждает, что классы должны использоваться повторно все вместе. Большинство причин, по которым классы должны объединяться в пакет, проистекают из зависимостей между классами, к которым я сейчас и перехожу.

Пакеты и зависимости

Диаграмма пакетов (package diagram) показывает пакеты и зависимости между ними. Я ввел понятие зависимости на *стр. 74*. При наличии пакетов для классов представления и пакетов для классов предметной области пакет представления зависит от пакета предметной области, если любой класс пакета представления зависит от какого-либо класса пакета предметной области. Таким образом, межпакетная зависимость обобщает зависимости между их содержимым.

В языке UML имеются разнообразные виды зависимостей, каждая 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. В средних и больших по размеру системах рисование диаграммы пакетов может быть одним из самых ценных приемов, позволяющих управлять их многомерной структурой. В идеале такая диаграмма должна быть сгенерирована на основании собственно исходного кода, так чтобы она реально отражала происходящее в системе.

Хорошая структура пакетов имеет прозрачный поток (clear flow) зависимостей – концепцию, которую трудно определить, но легко распознать. На рис. 7.2 показана простая диаграмма пакетов для промышленного приложения, которая хорошо структурирована и имеет прозрачный поток зависимостей.

Часто можно идентифицировать прозрачный поток, поскольку все зависимости идут в одном направлении. Это хороший индикатор правильно структурированной системы, но пакеты `data mapper` (преобразователь данных) на рис. 7.2 представляют исключение из этого эмпирического правила. Пакеты преобразователей данных действуют в качестве изолирующего уровня между пакетами предметной области

(domain) и пакетами базы данных (database) и служат примером шаблона Mapper (Преобразователь) [19].

Многие авторы утверждают, что в зависимостях не должно быть циклов (Acyclic Dependency Principle – принцип ацикличности зависимостей, [30]). Я не считаю это правило абсолютным, но думаю, что циклы необходимо локализовать, в частности не должно быть циклов, пересекающих уровни.

Чем больше зависимостей входит в пакет, тем более стабильным должен быть его интерфейс, поскольку любые изменения интерфейса отразятся на всех пакетах, зависящих от него (Stable Dependencies Principle – принцип стабильных зависимостей, [30]). Поэтому на рис. 7.2 пакету asset domain (предметная область собственности) требуется более стабильный интерфейс, чем пакету leasing data mapper (преобразователь данных аренды). Вы увидите, что зачастую более стабильные пакеты содержат относительно больше интерфейсов и абстрактных классов (Stable Abstractions Principle – принцип стабильных абстракций, [30]).

Отношения зависимостей не транзитивны (*стр. 75*). Чтобы убедиться в важности этого свойства для зависимостей, взгляните снова на рис. 7.2. Если изменяется класс пакета предметной области собственности, то может измениться и класс из пакета предметной области аренды (leasing domain). Но эти изменения не обязательно пройдут через представление аренды (leasing presentation). (Они проходят, толь-

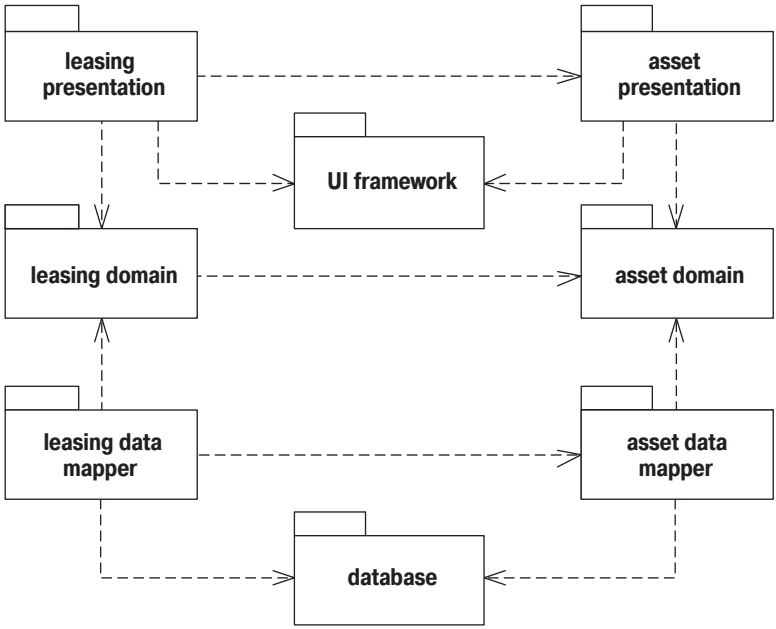

Рис. 7.2. *Диаграмма пакетов для промышленного предприятия*

ко если предметная область аренды изменяет свой интерфейс.) Некоторые пакеты используются в таком количестве мест, что можно запутаться в линиях зависимостей, идущих к ним. В таком случае действует соглашение, в соответствии с которым на пакете указывается ключевое слово «global» (глобальный).

В пакетах UML определены также конструкции, позволяющие пакетам импортировать классы из одного пакета в другой и объединять их, используя зависимости с ключевыми словами для их обозначения. Однако правила для такого типа элементов существенно отличаются в различных языках программирования. В целом, я думаю, что на практике значительно более полезна основная нотация зависимостей.

Аспекты пакетов

Рассматривая рис. 7.2, вы понимаете, что на диаграмме присутствуют структуры двух видов. Одна – структура уровня приложения: представление (presentation), предметная область (domain), преобразователь данных (data mapper) и база данных (database). Другая – структура предметных областей: аренда (leasing) и собственность (asset).

Это можно сделать более прозрачным, если разделить два данных аспекта так, как это сделано на рис. 7.3. На представленной диаграмме можно ясно увидеть каждый аспект. Однако эти два аспекта не явля-

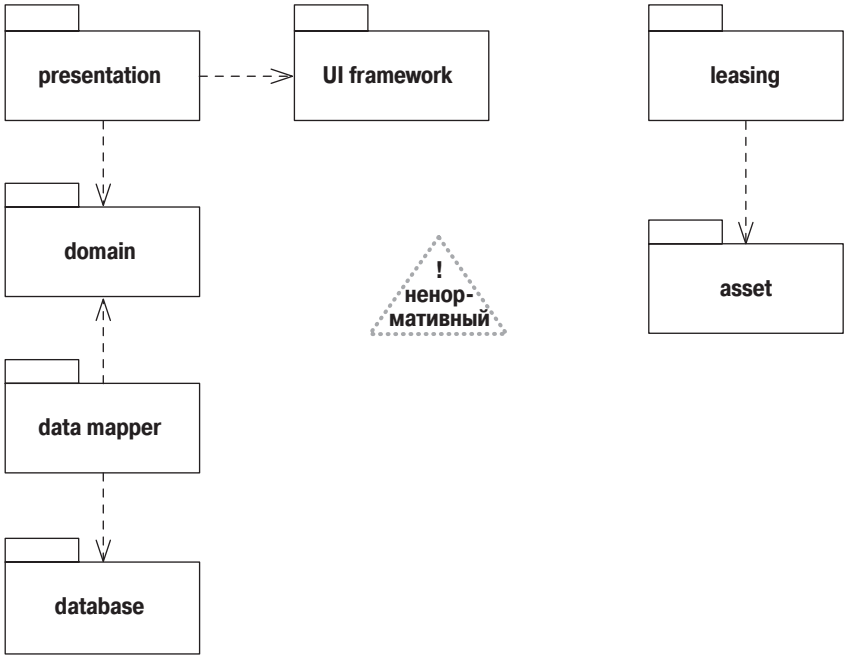

Рис. 7.3. *Разделение рис. 7.2 на два аспекта*

ются истинными пакетами, поскольку рассматриваемые классы можно объединить в один пакет. (Возможно, вам придется извлечь по одному классу из каждого аспекта.) Эта проблема является отражением проблемы иерархических пространств имен в языках программирования. Хотя диаграммы, подобные представленным на рис 7.3, не входят в стандарт языка UML, они зачастую очень удобны для объяснения структуры сложных приложений.

Реализация пакетов

Часто встречается ситуация, когда один пакет определяет интерфейс, который может быть реализован многими другими пакетами, как это показано на рис. 7.4. В данном случае отношение реализации означает, что шлюз базы данных (`Database Gateway`) определяет интерфейс, а другие классы шлюзов обеспечивают реализацию. На практике это может означать, что пакет шлюза базы данных (`Database Gateway`) содержит интерфейсы и абстрактные классы, которые полностью реализуются в других пакетах.

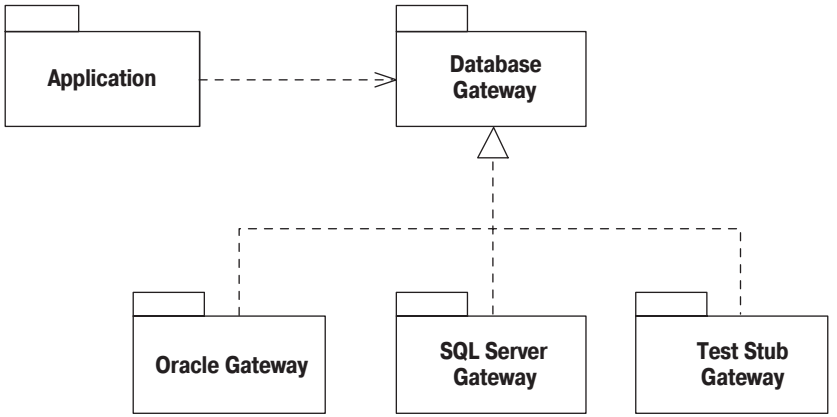

Рис. 7.4. *Пакет, реализованный другими пакетами*

Общепринято размещать интерфейс и его реализацию в разных пакетах. Действительно, клиентский пакет часто содержит интерфейс, который должен быть реализован другим пакетом (такую же нотацию затребованного интерфейса я обсуждал на *стр. 97*).

Допустим, что вы хотите предоставить некоторый пользовательский интерфейс (UI) выключателей (controls) для включения и выключения некоторого объекта. Мы хотим, чтобы он работал с различными устройствами, такими как обогреватели (heaters) или лампы (lights). Выключатели UI должны вызывать методы обогревателя, но мы не хотим, чтобы выключатели зависели от обогревателя. Мы можем избежать этой зависимости, определяя в пакете выключателей интерфейс,

который затем реализуется каждым классом, работающим с этими выключателями, как показано на рис. 7.5. Здесь представлен пример шаблона Separated Interface (Разделенный интерфейс) [19].

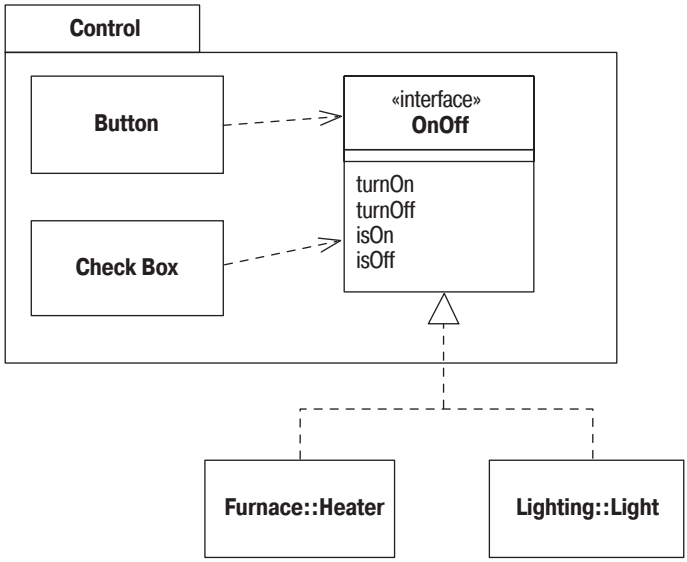

Рис. 7.5. *Определение затребованного интерфейса в клиентском пакете*

Когда применяются диаграммы пакетов

Я считаю, что диаграммы пакетов исключительно удобны в больших по размерам системах для представления картины зависимостей между основными элементами системы. Такие диаграммы хорошо соответствуют общепринятым программным структурам. Рисование диаграмм пакетов и зависимостей помогает держать под контролем зависимости приложения. Диаграммы пакетов представляют группирующий механизм времени компиляции. Для представления компоновки объектов во время выполнения применяются диаграммы составных структур (composite structure) (*стр. 155*).

Где найти дополнительную информацию

Лучшее описание пакетов и их использования, которое мне известно, – это книга Мартина [30]. Роберт Мартин почти патологически одержим зависимостями и описывает способы обращения с зависимостями, которые позволяют их минимизировать.

8

Диаграммы развертывания

Диаграммы развертывания представляют физическое расположение системы, показывая, на каком физическом оборудовании запускается та или иная составляющая программного обеспечения. Диаграммы развертывания очень просты, поэтому будем кратки.

На рис. 8.1 показан пример простой диаграммы развертывания. Главными элементами диаграммы являются узлы, связанные информационными путями. **Узел** (node) – это то, что может содержать программное обеспечение. Узлы бывают двух типов. **Устройство** (device) – это физическое оборудование: компьютер или устройство, связанное с системой. **Среда выполнения** (execution environment) – это программное обеспечение, которое само может включать другое программное обеспечение, например операционную систему или процесс-контейнер.

Узлы могут содержать **артефакты** (artifacts), которые являются физическим олицетворением программного обеспечения; обычно это файлы. Такими файлами могут быть исполняемые файлы (такие как файлы *.exe*, двоичные файлы, файлы DLL, файлы JAR, сборки или сценарии) или файлы данных, конфигурационные файлы, HTML-документы и т. д. Перечень артефактов внутри узла указывает на то, что на данном узле артефакт разворачивается в запускаемую систему.

Артефакты можно изображать в виде прямоугольников классов или перечислять их имена внутри узла. Если вы показываете эти элементы в виде прямоугольников классов, то можете добавить значок документа или ключевое слово «`artifact`». Можно сопровождать узлы или артефакты значениями в виде меток, чтобы указать различную интересную информацию об узле, например поставщика, операционную систему, местоположение – в общем, все, что придет вам в голову.

Часто у вас будет множество физических узлов для решения одной и той же логической задачи. Можно отобразить этот факт, нарисовав множество прямоугольников узлов или поставив число в виде значения-метки. На рис. 8.1 я обозначил три физических веб-сервера с по-

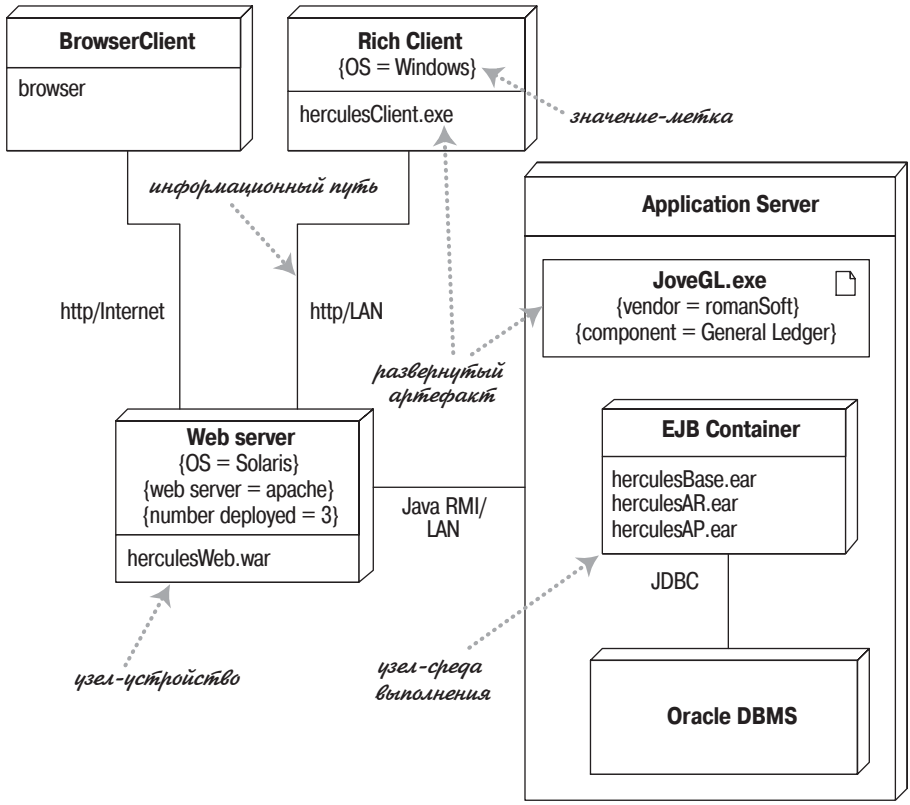

Рис. 8.1. *Пример диаграммы развертывания*

мощью метки `number deployed` (количество развернутых), но это не стандартная метка.

Артефакты часто являются реализацией компонентов. Это можно показать, задав значения-метки внутри прямоугольников артефактов.

Информационные пути между узлами представляют обмен информацией в системе. Можно сопровождать эти пути информацией об используемых информационных протоколах.

Когда применяются диаграммы развертывания

Пусть краткость этой главы не заставляет вас думать, что диаграммы развертывания можно не использовать. С их помощью очень удобно показывать размещение элементов, поэтому в случае любого нетривиального развертывания они могут оказаться очень полезными.

9

Прецеденты

Прецеденты – это технология определения функциональных требований к системе. Работа прецедентов заключается в описании типичных взаимодействий между пользователями системы и самой системой и предоставлении описания процесса ее функционирования. Вместо того чтобы описывать прецеденты в лоб, я предпочитаю подкрасться к ним сзади и начать с описания сценариев. **Сценарий** (scenario) – это последовательность шагов, описывающих взаимодействие пользователя и системы. Поэтому при наличии онлайнового магазина, основанного на веб-сайте, мы можем использовать сценарий «Покупка товара» (Buy a Product), в котором происходит следующее.

> *Покупатель просматривает каталог и помещает выбранные товары в корзину. При желании оплатить покупку он вводит информацию о кредитной карте и производит платеж. Система проверяет авторизацию кредитной карты и подтверждает оплату товара тотчас же и по электронной почте.*

Подобный сценарий описывает только одну ситуацию, которая может иметь место. Однако если авторизация кредитной карты окажется неудачной, то подобная ситуация может послужить предметом уже другого сценария. В другом случае у вас может быть постоянный клиент, для которого проверка информации о покупке и кредитной карте не обязательна, и это будет третий сценарий.

Так или иначе, но все эти сценарии похожи. Суть в том, что во всех трех сценариях у пользователя одна и та же цель: купить товар. Пользователь не всегда может это сделать, но цель остается. Именно цель пользователя является ключом к прецедентам: прецедент представляет собой множество сценариев, объединенных некоторой общей целью пользователя.

В терминах прецедента пользователи называются актерами. **Актер** (actor) представляет собой некую роль, которую пользователь играет по отношению к системе. Актерами могут быть пользователь, торговый представитель пользователя, менеджер по продажам и товаровед.

Актеры действуют в рамках прецедентов. Один актер может выполнять несколько прецедентов; и наоборот, в соответствии с одним прецедентом могут действовать несколько актеров. Обычно клиентов много, поэтому роль клиента могут играть многие люди. К тому же один человек может играть несколько ролей, например менеджер по продажам, выполняющий роль торгового представителя клиента. Актер не обязательно должен быть человеком. Если система предоставляет некоторый сервис другой компьютерной системе, то другая система является актером.

На самом деле *актер* – не совсем верный термин; возможно, термин *роль* (*role*) подошел бы лучше. Очевидно, имел место неправильный перевод со шведского языка, и в результате сообщество пользователей прецедентов теперь употребляет термин *актер*.

Прецеденты считаются важной частью языка UML. Однако удивительно то, что определение прецедентов в UML довольно скудное. В UML ничего не говорится о том, как определять содержимое прецедента. Все, что описано в UML, – это диаграмма прецедентов, которая показывает, как прецеденты связаны друг с другом. Но почти вся ценность прецедентов как раз в их содержании, а диаграмма имеет ограниченное значение.

Содержимое прецедентов

Не существует стандартного способа описания содержимого прецедента; в разных случаях применяются различные форматы. На рис. 9.1 показан общий стиль использования. Вы начинаете с выбора одного из сценариев в качестве **главного успешного сценария** (main success scenario). Сначала вы описываете тело прецедента, в котором главный успешный сценарий представлен последовательностью нумерованных шагов. Затем берете другой сценарий и вставляете его в виде **расширения** (extension), описывая его в терминах изменений главного успешного сценария. Расширения могут быть успешными – пользователь достиг своей цели, как в варианте 3а, или неудачными, как в варианте 6а.

В каждом прецеденте есть ведущий актер, который посылает системе запрос на обслуживание. Ведущий актер – это актер, желание которого пытается удовлетворить прецедент и который обычно, но не всегда, является инициатором прецедента. Одновременно могут быть и другие актеры, с которыми система также взаимодействует во время выполнения прецедента. Они называются второстепенными актерами.

Каждый шаг в прецеденте – это элемент взаимодействия актера с системой. Каждый шаг должен быть простым утверждением и должен четко указывать, кто выполняет этот шаг. Шаг должен показывать намерение актера, а не механику его действий. Следовательно, в прецеденте интерфейс актера не описывается. Действительно, составление прецедента обычно предшествует разработке интерфейса пользователя.

Покупка товара
Целевой уровень: уровень моря

Главный успешный сценарий:
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 подтверждает оплату товаров немедленно.
8. Система посылает подтверждение оплаты товаров по адресу электронной почты покупателя.

Расширения:
3а. Клиент является постоянным покупателем.
.1: Система предоставляет информацию о текущей покупке и ее цене, а также информацию о счете.
.2: Покупатель может согласиться или изменить значения по умолчанию, затем возвращаемся к шагу 6 главного успешного сценария.
6а. Система не подтверждает авторизацию счета.
.1: Пользователь может повторить ввод информации о кредитной карте или закончить сеанс.

Рис. 9.1*. Пример текста прецедента*

Расширение внутри прецедента указывает условие, которое приводит к взаимодействиям, отличным от описанных в главном успешном сценарии (main success scenario, MSS), и устанавливает, в чем состоят эти отличия. Расширение начинается с имени шага, на котором определяется это условие, и предоставляет краткое описание этого условия. Следуйте этому условию, нумеруя шаги таким же образом, что и в главном успешном сценарии. Заканчивайте эти шаги описанием точки возврата в главный успешный сценарий, если это необходимо.

Структура прецедента – это отличный инструмент для поиска альтернатив главного успешного сценария. На каждом шаге спрашивайте: «Что может еще произойти?» и в частности «Что может пойти не так?» Обычно лучше сначала изучить все возможные условия расширения, чтобы потом не увязнуть в трясине работы над последствиями. Таким образом, вы, возможно, обдумаете больше условий, что приведет к меньшему количеству ошибок, которые потом пришлось бы отлавливать.

Сложный шаг в прецеденте можно представить другим прецедентом. В терминах языка UML мы говорим, что первый прецедент **включает** (includes) второй. Не существует стандартного способа показать в тексте

включение прецедента, но я думаю, что подчеркивание, которое предполагает гиперссылку, работает прекрасно, а во многих инструментах действительно будет гиперссылкой. Так, на рис. 9.1 первый шаг включает шаблон «просматривает каталог и выбирает товары для покупки».

Включенные прецеденты могут быть полезными в случае сложных шагов, которые иначе загромождали бы главный сценарий, или когда одни и те же шаги присутствуют в нескольких сценариях. Однако не пытайтесь разбивать прецеденты на подпрецеденты и использовать их для функциональной декомпозиции. Такая декомпозиция – хороший способ потерять много времени.

Наряду с шагами сценария можно вставить в прецедент дополнительную общую информацию.

- **Предусловие** (pre-condition) описывает действия, обязательно выполняемые системой перед тем, как она разрешит начать работу прецедента. Это полезная информация, позволяющая разработчикам не проверять некоторые условия в их программе.
- **Гарантия** (guarantee) описывает обязательные действия системы по окончании работы шаблона ответа. Успешные гарантии выполняются после успешного сценария; минимальные гарантии выполняются после любого сценария.
- **Триггер** (trigger) определяет событие, инициирующее выполнение прецедента.

При рассмотрении дополнительных элементов относитесь к этому скептически. Лучше сделать слишком мало, чем слишком много. Кроме того, приложите максимум усилий, чтобы сделать прецедент кратким и легким для чтения. Я убедился, что излишне подробный прецедент, который трудно читать, скорее приведет к провалу, чем к достижению цели. Не обязательно записывать все детали; устное общение часто бывает очень эффективным, особенно во время итеративного цикла, когда необходимые условия быстро выполняются запущенной программой.

Степень детализации, необходимая в прецеденте, зависит от уровня риска этого прецедента. Часто детали нужны в начале только немногих ключевых прецедентов, другие можно конкретизировать непосредственно перед их реализацией.

Диаграммы прецедентов

Я уже говорил, что язык UML умалчивает о содержимом прецедента, но предоставляет формат диаграммы, позволяющий его отображать (рис. 9.2). Хотя диаграмма иногда оказывается полезной, без нее можно обойтись. При разработке прецедента не стоит прилагать много усилий для создания диаграммы. Вместо этого лучше сконцентрироваться на текстовом содержании прецедентов.

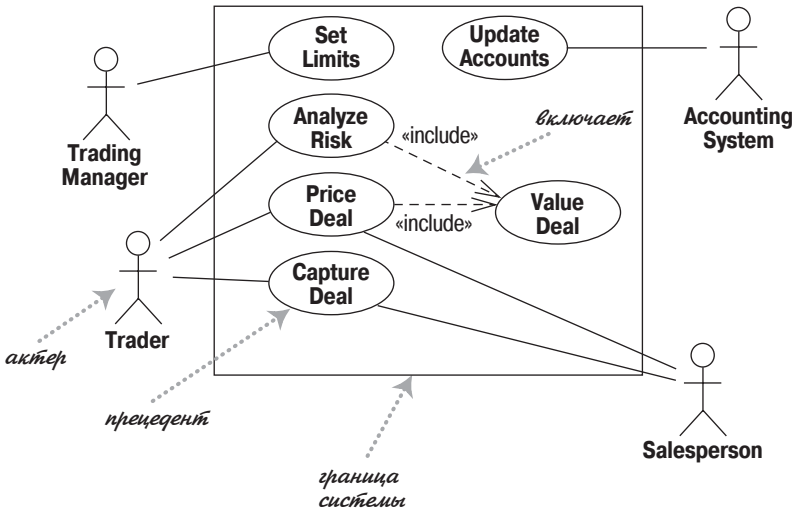

Рис. 9.2. *Диаграмма прецедентов*

Лучше всего обдумывать диаграмму прецедентов с помощью графической таблицы, показывающей их содержимое. Она напоминает диаграмму контекста, используемую в структурных методах, поскольку она показывает границы системы и ее взаимодействие с внешним миром. Диаграмма прецедентов показывает актеров, прецеденты и отношения между ними:

- Какие актеры выполняют тот или иной прецедент
- Какие прецеденты включают другие прецеденты

В языке UML помимо отношения «include» (включает) есть и другие типы отношений между прецедентами, например отношение «extend» (расширяет). Я настоятельно рекомендую его избегать. Слишком часто на моих глазах разработчики целыми командами надолго погружались в рассмотрение различных отношений между прецедентами, понапрасну растрачивая силы. Лучше уделяйте больше внимания текстовому описанию прецедента; именно в этом заключается истинная ценность этой технологии.

Уровни прецедентов

Общая проблема прецедентов состоит в том, что, увлекшись взаимодействием пользователя с системой, можно не обратить внимание на тот факт, что лучшим способом решения проблемы может быть изменение самого бизнес-процесса. Часто можно слышать упоминание о прецедентах системы и прецедентах бизнес-процессов. Конечно, эта терминология не является точной, но обычно считается, что **прецедент системы** (system use case) описывает особенности взаимодействия с программ-

ным обеспечением, тогда как **прецедент бизнес-процесса** (business use case) представляет собой реакцию бизнес-процесса на действие клиента или некоторое событие.

В книге Кокборна [10] предлагается схема уровней прецедентов. Базовый прецедент находится на «уровне моря». Прецеденты **уровня моря** (sea level) обычно представляют отдельное взаимодействие ведущего актера и системы. Такие прецеденты предоставляют ведущему актеру какой-либо полезный результат и обычно занимают от пары минут до получаса. Прецеденты, которые существуют в системе, только если они включены в прецеденты уровня моря, называются прецедентами **уровня рыб** (fish level). Прецеденты высшего уровня, **уровня воздушного змея** (kite-level), показывают, как прецеденты уровня моря настраиваются на более широкое взаимодействие с бизнес-процессами. Обычно прецеденты уровня воздушного змея являются прецедентами бизнес-процессов, а на уровне моря и на уровне рыб находятся прецеденты системы. Большинство ваших прецедентов должно принадлежать уровню моря. Я предпочитаю указывать уровень в начале прецедента, как показано на рис. 9.1.

Прецеденты и возможности (или пожелания)

Во многих подходах возможности системы применяются для описания требований к системе; в экстремальном программировании (Extreme Programming) возможности системы называются пожеланиями пользователя. Общим является вопрос о том, как установить соответствие между возможностями и прецедентами.

Использование возможностей – это хороший способ разделения системы на блоки при планировании итеративного процесса, в результате чего каждая итерация предоставляет определенное количество возможностей. Отсюда следует, что хотя оба приема описывают требования, их цели различны.

Описать возможности можно сразу, но многие специалисты считают более удобным в первую очередь разработать прецеденты, а уже затем сгенерировать список возможностей. Возможность может быть представлена целым прецедентом, сценарием в прецеденте, шагом в прецеденте или каким-либо вариантом поведения, например добавление еще одного метода вычисления амортизации при оценке вашего имущества, который не указан в описании прецедента. Обычно возможности получаются более четко определенными, чем прецеденты.

Когда применяются прецеденты

Прецеденты представляют собой ценный инструмент для понимания функциональных требований к системе. Первый вариант прецедентов должен составляться на ранней стадии выполнения проекта. Более

подробные версии прецедентов должны появляться непосредственно перед реализацией данного прецедента.

Важно помнить, что прецеденты представляют взгляд на систему *со стороны*. А раз так, то не ждите какого-либо соответствия между прецедентами и классами внутри системы.

Чем больше прецедентов я вижу, тем менее ценной мне кажется диаграмма прецедентов. Несмотря на то что в языке UML ничего не говорится о тексте прецедентов, именно текстовое содержание прецедентов является основной ценностью этой технологии.

Большая опасность прецедентов заключается в том, что разработчики делают их очень сложными и застревают на них. Обычно чем меньше вы делаете, тем меньший вред можете нанести. Если у вас немного информации, то получится короткий, легко читаемый документ, который явится отправной точкой для вопросов. Если информации слишком много, то вряд ли кто-то вообще будет ее изучать и пытаться понять.

Где найти дополнительную информацию

Впервые прецеденты были изложены в доступной форме Айваром Джекобсоном в [24].

Хотя прецеденты существовали и до этого, в упомянутой публикации была предпринята попытка некоторой стандартизации их использования. В языке UML ничего не говорится о такой важной вещи, как содержимое прецедентов, в нем стандартизован только менее важный элемент – диаграмма прецедентов. Поэтому можно услышать целый ряд различных мнений о прецедентах.

Однако за последние несколько лет книга Кокборна [10] стала стандартом по этому предмету. Я следовал терминологии и советам этой книги по той веской причине, что когда в прошлом мы расходились во мнениях, в конце концов я соглашался с Алистером Кокборном. Он поддерживает веб-сайт по адресу *http://usecases.org*. В книге [11] предлагается убедительный процесс получения пользовательских интерфейсов из прецедентов; посетите также сайт по адресу *http://foruse.com*.

10

Диаграммы состояний

Диаграммы состояний (state machine diagrams) – это известная технология описания поведения системы. В том или ином виде диаграммы состояний существуют с 1960 года, и на заре объектно-ориентированного программирования они применялись для представления поведения системы. В объектно-ориентированных подходах вы рисуете диаграмму состояний единственного класса, чтобы показать поведение одного объекта в течение его жизни.

Всякий раз, когда пишут о конечных автоматах, в качестве примеров неизбежно приводят системы круиз-контроля или торговые автоматы. Поскольку они мне слегка наскучили, я решил использовать контроллер секретной панели управления в Готическом замке. В этом замке я хочу так спрятать свои сокровища, чтобы их было трудно найти. Для того чтобы получить доступ к замку сейфа, я должен вытащить из канделябра стратегическую свечу, но замок появится, только если дверь закрыта. После появления замка я могу вставить в него ключ и открыть сейф. Для дополнительной безопасности я сделал так, чтобы сейф можно было открыть только после извлечения свечи. Если вор не обратит внимания на эту предосторожность, то я спущу с цепи отвратительного монстра, который проглотит вора.

На рис. 10.1 показана диаграмма состояний класса контроллера, который управляет моей необычной системой безопасности. Диаграмма состояния начинается с состояния создаваемого объекта контроллера: состояния `Wait` (Ожидание). На диаграмме это обозначено с помощью **начального псевдосостояния** (initial pseudostate), которое не является состоянием, но имеет стрелку, указывающую на начальное состояние.

На диаграмме показано, что контроллер может находиться в одном из трех состояний: `Wait` (Ожидание), `Lock` (Замок) и `Open` (Открыт). На диаграмме также представлены правила, согласно которым контроллер переходит из одного состояния в другое. Эти правила представлены в виде переходов – линий, связывающих состояния.

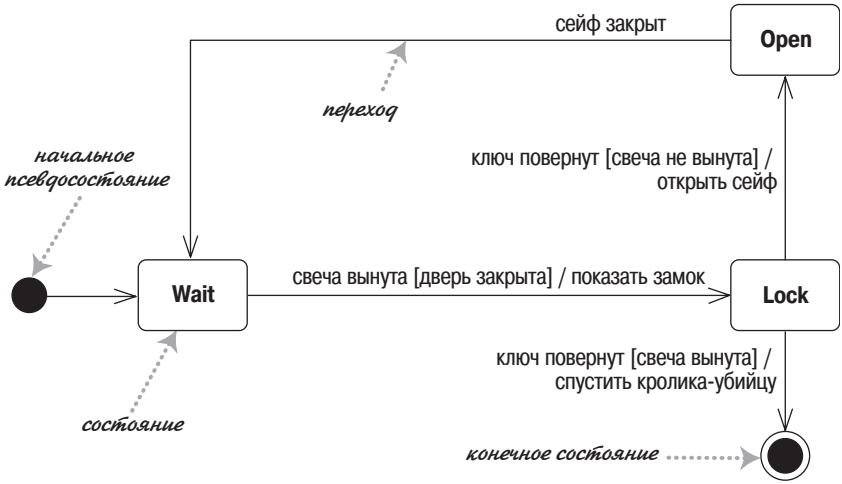

Рис. 10.1. *Простая диаграмма состояний*

Переход (transition) означает перемещение из одного состояния в другое. Каждый переход имеет свою метку, которая состоит из трех частей: `триггер-идентификатор [защита]/активность` (`trigger-signature [guard]/activity`). Все они не обязательны. Как правило, `триггер-идентификатор` – это единственное событие, которое может вызвать изменение состояния. `Защита`, если она указана, представляет собой логическое условие, которое должно быть выполнено, чтобы переход имел место. `Активность` – это некоторое поведение системы во время перехода. Это может быть любое поведенческое выражение. Полная форма `триггера-идентификатора` может включать несколько событий и параметров. Переход из состояния `Wait` (рис. 10.1) в другое состояние можно прочесть как «В состоянии `Wait`, если свеча удалена, вы видите замок и переходите в состояние `Lock`».

Все три части описания перехода не обязательны. Пропуск активности означает, что в процессе перехода ничего не происходит. Пропуск защиты означает, что переход всегда осуществляется, если происходит инициирующее событие. Триггер-идентификатор отсутствует редко, но и так бывает. Это означает, что переход происходит немедленно, что можно наблюдать главным образом в состояниях активности, о чем я расскажу в нужный момент.

Когда в определенном состоянии происходит событие, то из этого состояния можно совершить только один переход, например в состоянии `Lock` (рис. 10.1) защиты должны быть взаимно исключающими. Если событие происходит, а разрешенных переходов нет – например закрытие сейфа в состоянии `Wait` или удаление свечи при открытой двери, – событие игнорируется.

Конечное состояние (final state) означает, что конечный автомат закончил работу, что вызывает удаление объекта контроллера. Так что

для тех, кто имел неосторожность попасть в мою ловушку, сообщаю, что поскольку объект контроллера прекращает свое существование, я вынужден посадить кролика обратно в клетку, вымыть пол и перегрузить систему.

Помните, что конечные автоматы могут показывать только те объекты, которые непосредственно наблюдаются или действуют. Поэтому, хотя вы могли ожидать, что я положу что-нибудь в сейф или что-нибудь возьму оттуда, когда дверь открыта, я не отметил это на диаграмме, поскольку контроллер об этом ничего сообщить не может.

Когда разработчики говорят об объектах, они часто ссылаются на состояние объектов, имея в виду комбинацию всех данных, содержащихся в полях объектов. Однако состояние на диаграмме конечного автомата является более абстрактным понятием состояния; суть в том, что различные состояния предполагают различные способы реакции на события.

Внутренние активности

Состояния могут реагировать на события без совершения перехода, используя **внутренние активности** (internal activities), и в этом случае событие, защита и активность размещаются внутри прямоугольника состояния.

На рис. 10.2 представлено состояние с внутренними активностями символов и событиями системы помощи, которые вы можете наблюдать в текстовых полях редактора UI. Внутренняя активность подобна **самопереходу** (self-transition) – переходу, который возвращает в то же самое состояние. Синтаксис внутренних активностей построен по той же логической схеме события, защиты и процедуры.

На рис. 10.2 показаны также специальные активности: входная и выходная активности. **Входная активность** выполняется всякий раз, когда вы входите в состояние; **выходная активность** – всякий раз, когда вы покидаете состояние. Однако внутренние активности не инициируют входную и выходную активности; в этом состоит различие между внутренними активностями и самопереходами.

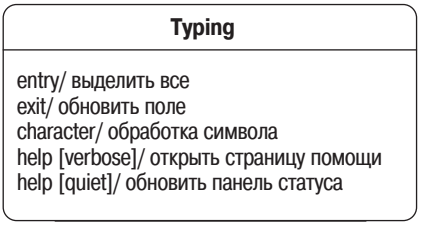

Рис. 10.2. *Внутренние события, показанные в состоянии набора текста в текстовом поле*

Состояния активности

В состояниях, которые я описывал до сих пор, объект молчит и ожидает следующего события, прежде чем что-нибудь сделать. Однако возможны состояния, в которых объект проявляет некоторую активность.

Состояние `Searching` (Поиск) на рис. 10.3 является таким **состоянием активности** (activity state): ведущаяся активность обозначается символом `do/`; отсюда термин **do-activity** (проявлять активность). После того как поиск завершен, выполняются переходы без активности, например показ нового оборудования (`Display New Hardware`). Если в процессе активности происходит событие отмены (cancel), то do-активность просто прерывается и мы возвращаемся в состояние `Update Hardware Window` (Обновление окна оборудования).

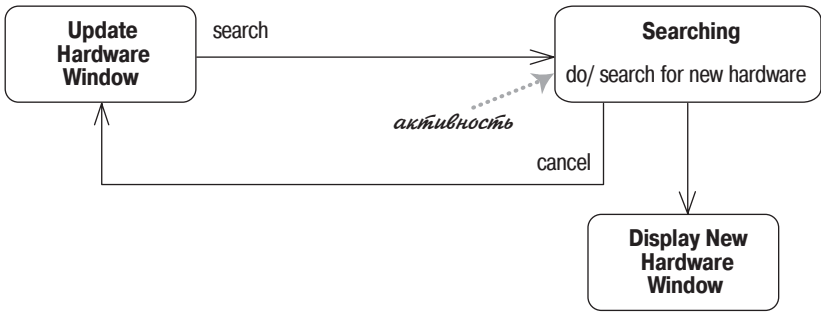

Рис. 10.3. *Состояние с активностью*

И do-активности, и обычные активности представляют проявление некоторого поведения. Решающее различие между ними заключается в том, что обычные активности происходят «мгновенно» и не могут быть прерваны обычными событиями, тогда как do-активности могут выполняться в течение некоторого ограниченного времени и могут прерываться, как показано на рис. 10.3. Мгновенность для разных систем трактуется по-разному; для систем реального времени это может занимать несколько машинных инструкций, а для настольного программного обеспечения может составить несколько секунд.

В UML 1 обычные активности обозначались термином **action** (действие), а термин **activity** (активность) применялся только для do-активностей.

Суперсостояния

Часто бывает, что несколько состояний имеют общие переходы и внутренние активности. В таких случаях можно их превратить в подсостояния (substates), а общее поведение перенести в суперсостояние (superstate), как показано на рис. 10.4. Без суперсостояния пришлось бы рисовать переход cancel (отмена) для всех трех состояний внутри состояния `Enter Connection Details` (Ввод подробностей соединения).

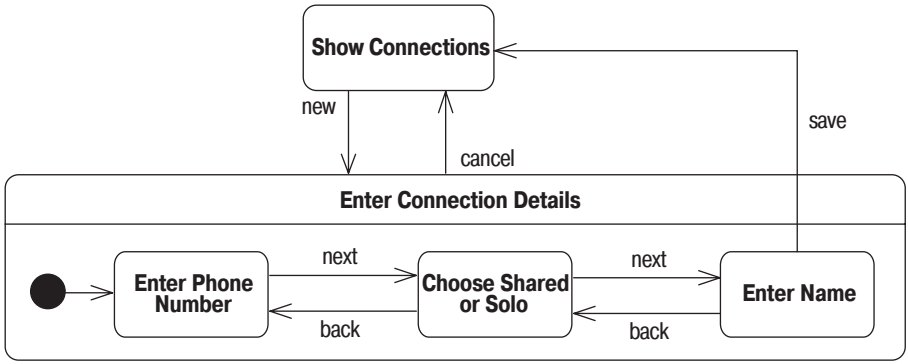

Рис. 10.4. Суперсостояние с вложенными подсостояниями

Параллельные состояния

Состояния могут быть разбиты на несколько параллельных состояний, запускаемых одновременно. На рис. 10.5 показан трогательно простой будильник, который может включать либо CD, либо радио и показывать либо текущее время, либо время сигнала.

Опции CD/радио и текущее время/время сигнала являются параллельными. Если бы вы захотели представить это с помощью диаграммы непараллельных состояний, то получилась бы беспорядочная диаграмма при необходимости добавить состояния. Разделение двух областей поведения на две диаграммы состояний делает ее значительно яснее.

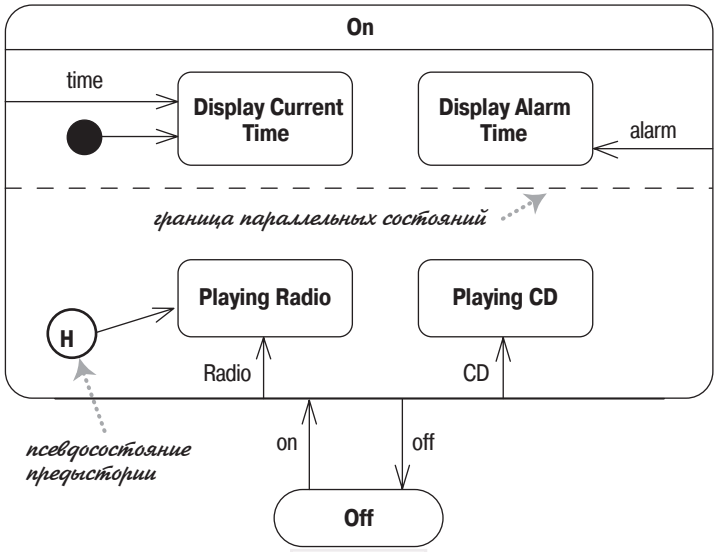

Рис. 10.5. Параллельные состояния

Рис. 10.5 включает также **состояние предыстории** (history pseudo-state). Это означает, что когда включены часы, опция радио/CD переходит в состояние, в котором находились часы, когда они были выключены. Стрелка, выходящая из предыстории, показывает, какое состояние существовало изначально, когда отсутствовала предыстория.

Реализация диаграмм состояний

Диаграмму состояний можно реализовать тремя основными способами: с помощью вложенного оператора switch, паттерна State и таблицы состояний. Самый прямой подход в работе с диаграммами состояний – это вложенный оператор switch, такой как на рис. 10.6.

```
public void HandleEvent (PanelEvent anEvent) {
   switch (CurrentState) {
      case PanelState.Open :
         switch (anEvent) {
            case PanelEvent.SafeClosed :
               CurrentState = PanelState.Wait;
               break;
         }
         break;
      case PanelState.Wait :
         switch (anEvent) {
            case PanelEvent.CandleRemoved :
               if (isDoorOpen) {
                  RevealLock();
                  CurrentState = PanelState.Lock;
               }
              break;
         }
         break;
      case PanelState.Lock :
         switch (anEvent) {
            case PanelEvent.KeyTurned :
               if (isCandleIn) {
                  OpenSafe();
                  CurrentState = PanelState.Open;
               } else {
                  ReleaseKillerRabbit();
                  CurrentState = PanelState.Final;
               }
               break;
         }
         break;
     }
  }
}
```

***Рис. 10.6.** Вложенный оператор switch на языке C# для обработки перехода состояний, представленного на рис. 10.1*

Хотя этот способ и прямой, но очень длинный даже для этого простого случая. Кроме того, при данном подходе очень легко потерять контроль, поэтому я не люблю применять его даже в элементарных ситуациях.

Паттерн «Состояние» (State pattern) [21] представляет иерархию классов состояний для обработки поведения состояний. Каждое состояние на диаграмме имеет свой подкласс состояния. Контроллер имеет методы для каждого события, которые просто перенаправляют к классу состояния. Диаграмма состояний, показанная на рис. 10.1, могла бы быть реализована с помощью классов, представленных на рис. 10.7.

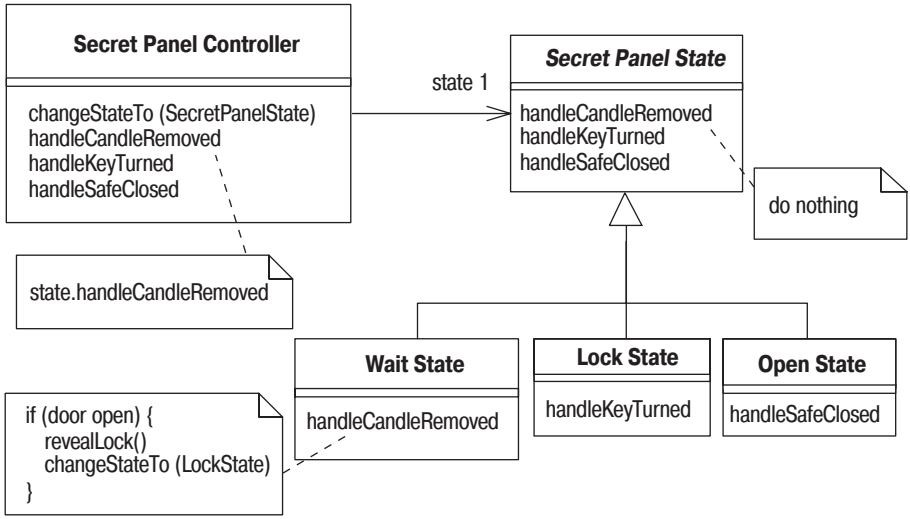

Рис. 10.7. *Паттерн «Состояние», реализующий диаграмму на рис. 10.1*

Вершиной иерархии является абстрактный класс, который содержит описание всех методов, обрабатывающих события, но без реализации. Для каждого конкретного состояния достаточно переписать метод-обработчик определенного события, инициирующего переход из состояния.

Таблица состояний представляет диаграмму состояний в виде данных. Так, диаграмма на рис. 10.1 может быть представлена в виде табл. 10.1. Затем мы строим интерпретатор, который использует таблицу состояний во время выполнения программы, или генератор кода, который порождает классы на основе этой таблицы.

Очевидно, большая часть работы над таблицей состояний проводится однажды, но затем ее можно использовать всякий раз, когда надо решить проблему, связанную с состояниями. Таблица состояний времени выполнения может быть модифицирована без перекомпиляции, что в некотором смысле удобно. Шаблон состояний собрать легче, и хотя для каждого состояния требуется отдельный класс, но размер кода, который при этом надо написать, совсем невелик.

Таблица 10.1. Таблица состояний для диаграммы на рис. 10.1

Исходное состояние	Целевое состояние	Событие	Защита	Процедура
Wait	Lock	Candle removed (свеча удалена)	Door open (дверца открыта)	Reveal lock (показать замок)
Lock	Open	Key turned (ключ повернут)	Candle in (свеча на месте)	Open safe (открыть сейф)
Lock	Final	Key turned (ключ повернут)	Candle out (свеча удалена)	Release killer rabbit (освободить убийцу-кролика)
Open	Wait	Safe closed (сейф закрыт)		

Приведенные реализации практически минимальные, но они дают представление о том, как применять диаграммы состояний. В каждом случае реализация моделей состояний приводит к довольно стереотипной программе, поэтому обычно для этого лучше прибегнуть к тому или иному способу генерации кода.

Когда применяются диаграммы состояний

Диаграммы состояний хороши для описания поведения одного объекта в нескольких прецедентах. Но они не очень подходят для описания поведения, характеризующегося взаимодействием множества объектов. Поэтому имеет смысл совместно с диаграммами состояний применять другие технологии. Например, диаграммы взаимодействия (глава 4) прекрасно описывают поведение нескольких объектов в одном прецеденте, а диаграммы деятельности (глава 11) хороши для показа основной последовательности действий нескольких объектов в нескольких прецедентах.

Не все считают диаграммы состояний естественными. Понаблюдайте, как специалисты работают с ними. Вполне возможно, что члены вашей команды не думают, что диаграммы состояний подходят для их стиля работы. Это не самая большая трудность; вы должны не забывать совместно использовать различные приемы работы.

Если вы применяете диаграммы состояний, то не старайтесь нарисовать их для каждого класса системы. Такой подход часто применяется в целях формально строгой полноты, но почти всегда это напрасная трата сил. Применяйте диаграммы состояний только для тех классов, которые проявляют интересное поведение, когда построение диаграммы состояний помогает понять, как все происходит. Многие специалисты считают, что редактор UI и управляющие объекты имеют функциональные средства, полезные при отображении с помощью диаграммы состояний.

Где найти дополнительную информацию

И руководство пользователя по UML [6], и справочное руководство [40] содержат более подробную информацию о диаграммах состояний. Проектировщики систем реального времени предпочитают интенсивно применять модели состояний, поэтому неудивительно, что в книге Дугласа [15] много говорится о диаграммах состояний, включая информацию об их реализации. В книге Мартина [30] есть хорошая глава о различных способах реализации диаграмм состояний.

11

Диаграммы деятельности

Диаграммы деятельности – это технология, позволяющая описывать логику процедур, бизнес-процессы и потоки работ. Во многих случаях они напоминают блок-схемы, но принципиальная разница между диаграммами деятельности и нотацией блок-схем заключается в том, что первые поддерживают параллельное процессы.

Диаграммы деятельности подвергались самым большим изменениям при смене версий языка UML, поэтому неудивительно, что они были снова изменены и существенно расширены в UML 2. В UML 1 диаграммы деятельности рассматривались как особый случай диаграмм состояний. Это вызвало немало трудностей у специалистов, моделирующих потоки работ, для которых хорошо подходят диаграммы деятельности. В UML 2 это ограничение было ликвидировано.

На рис. 11.1 показан пример простой диаграммы деятельности. Мы стартуем с начального узла (initial node), а затем выполняем операцию `Receive Order` (Принять заказ). Затем идет ветвление (fork), которое имеет один входной поток и несколько выходных параллельных потоков.

Из рис. 11.1 видно, что операции `Fill Order` (Заполнить заявку), `Send Invoice` (Послать счет) и следующие за ними выполняются параллельно. По существу, в данном случае это означает, что последовательность операций не имеет значения. Я могу заполнить заявку, послать счет, доставить товар (`Delivery`), а затем получить оплату (`Receive Payment`); или я могу послать счет, получить оплату, заполнить заявку, а затем доставить товар. Посмотрите на рисунок.

Я могу также выполнять операции поочередно. Я беру со склада первую позицию заказа, печатаю счет, беру вторую позицию заказа, кладу счет в конверт и т. д. Или я могу что-то делать одновременно: печатать счет одной рукой, а другой рукой брать что-нибудь со склада. Согласно диаграмме, любая из этих последовательностей действий допустима.

Диаграмма деятельности позволяет любому, кто выполняет данный процесс, выбирать порядок действий. Другими словами, диаграмма

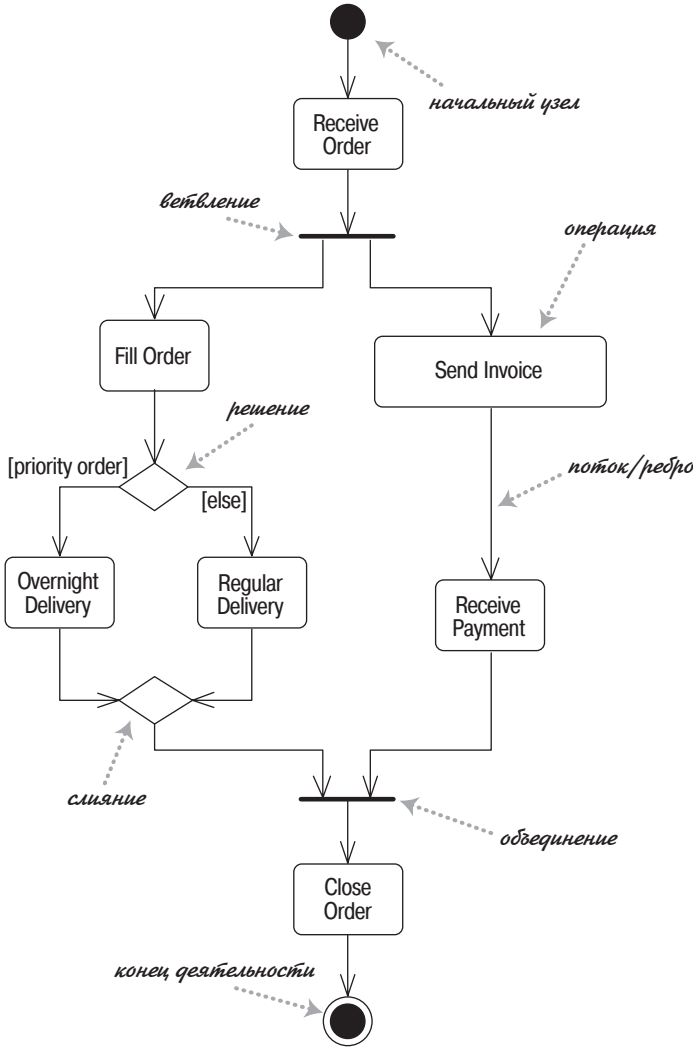

Рис. 11.1. *Простая диаграмма деятельности*

только устанавливает правила обязательной последовательности действий, которым я должен следовать. Это важно для моделирования бизнес-процессов, поскольку эти процессы часто выполняются параллельно. Такие диаграммы также полезны при разработке параллельных алгоритмов, в которых независимые потоки могут выполнять работу параллельно.

При наличии параллелизма необходима синхронизация. Мы не закрываем заказ, пока он не оплачен и не доставлен. Это показывается с помощью **объединения** (join) перед операцией `Close Order` (Закрыть заказ). Исходящий из объединения поток выполняется только в том слу-

чае, когда все входящие потоки достигли объединения. Поэтому вы можете закрыть заказ, только когда принята оплата и заказ доставлен.

В UML 1 действовали определенные правила для балансировки ветвлений и объединений, так как диаграммы деятельности представляли особый случай диаграмм состояний. В UML 2 в такой балансировке уже нет необходимости.

Заметьте, что узлы на диаграмме деятельности называются операциями (actions), а не активностями (activities). Строго говоря, деятельность относится к последовательности действий, поэтому диаграмма представляет деятельность, состоящую из операций.

Условное поведение схематически обозначается с помощью решений (decisions) и слияний (merges). **Решение**, которое в UML 1 называлось ветвью, имеет один входящий поток и несколько защищенных выходных потоков. Каждый выходной поток имеет защиту – условное выражение, помещенное в квадратные скобки. Каждый раз при достижении решения выбирается только один из выходных потоков, поэтому защиты должны быть взаимно исключающими. Применение `[else]` в качестве защиты означает, что поток `[else]` используется в том случае, когда другие защиты данного решения принимают ложное значение.

На рис. 11.1 решение располагается после операции заполнения заявки. Если у вас срочный заказ, то он выполняется в течение суток (`Overnight Delivery`); в противном случае производится обычная доставка (`Regular Delivery`).

Слияние (merge) имеет несколько входных потоков и один выходной. Слияние обозначает завершение условного поведения, которое было начато решением.

На моей диаграмме каждая операция имеет один входящий в нее поток и один выходящий. В UML 1 подразумевалось, что несколько входящих потоков имеют слияние. Другими словами, операция выполнялась, если запускался любой поток. В UML 2 это было изменено, так что вместо слияния предполагается объединение; таким образом, операция выполняется, только если все потоки пройдены. Поэтому я рекомендую применять операции с единственным входным потоком и единственным выходным, а также явно показывать все объединения и слияния; это избавит вас от путаницы.

Декомпозиция операции

Операции могут быть разбиты на вложенные деятельности (subactivities). Я могу взять алгоритм доставки, показанный на рис. 11.1, и определить его как самостоятельную деятельность (рис. 11.2), а затем вызвать его как операцию (рис. 11.3).

Операции могут быть реализованы или как вложенные деятельности или как методы классов. Вложенную деятельность можно обозначить

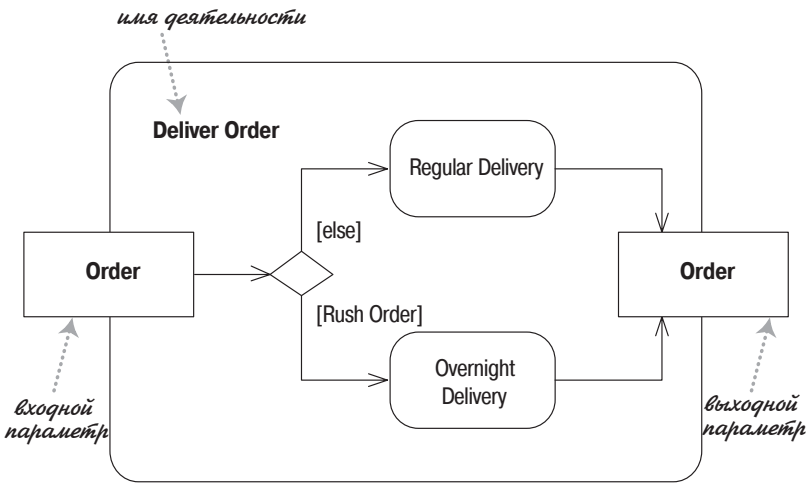

Рис. 11.2. *Дополнительная диаграмма деятельности*

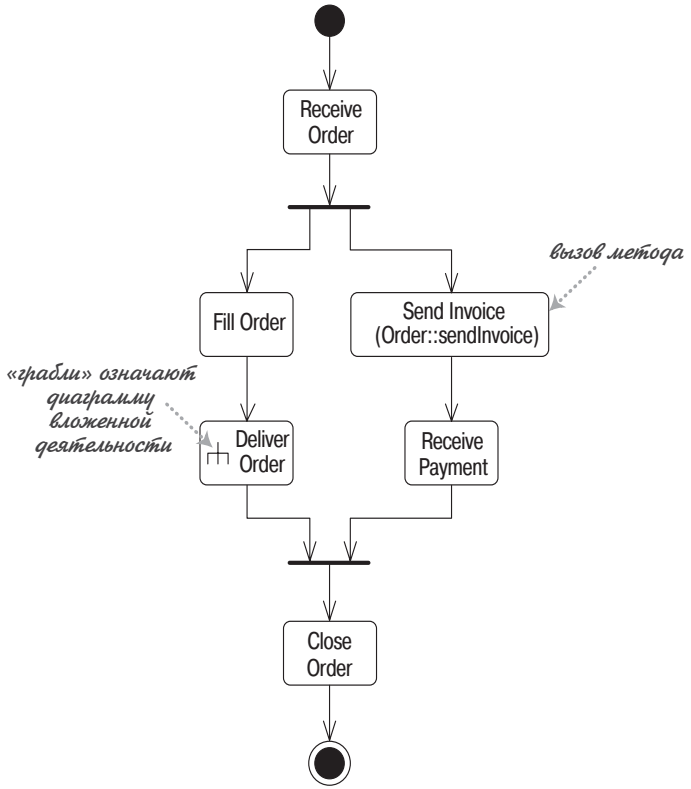

Рис. 11.3. *Деятельность из рис. 11.1 модифицирована для вызова деятельности из рис. 11.2*

с помощью символа «граблей». Вызов метода отображается с помощью синтаксиса `имя-класса::имя-метода`. Можно также вставить в символ операции фрагмент кода, если поведение представлено не единственным вызовом метода.

Разделы

Диаграммы деятельности рассказывают о том, что происходит, но ничего не говорят о том, кто какие действия выполняет. В программировании это означает, что диаграмма не отражает, какой класс является ответственным за ту или иную операцию. В моделировании бизнес-процессов это означает, что не отражено распределение обязанностей между подразделениями фирмы. Это не всегда представляет собой трудность; часто имеет смысл сконцентрироваться на том, что происходит, а не на том, кто какую роль играет в данном сценарии.

Можно разбить диаграмму деятельности на разделы (partitions), чтобы показать, кто что делает, то есть какие операции выполняет тот или иной класс или подразделение предприятия. На рис. 11.4 приведен простой пример, показывающий, как операции по обработке заказа могут быть распределены между различными подразделениями.

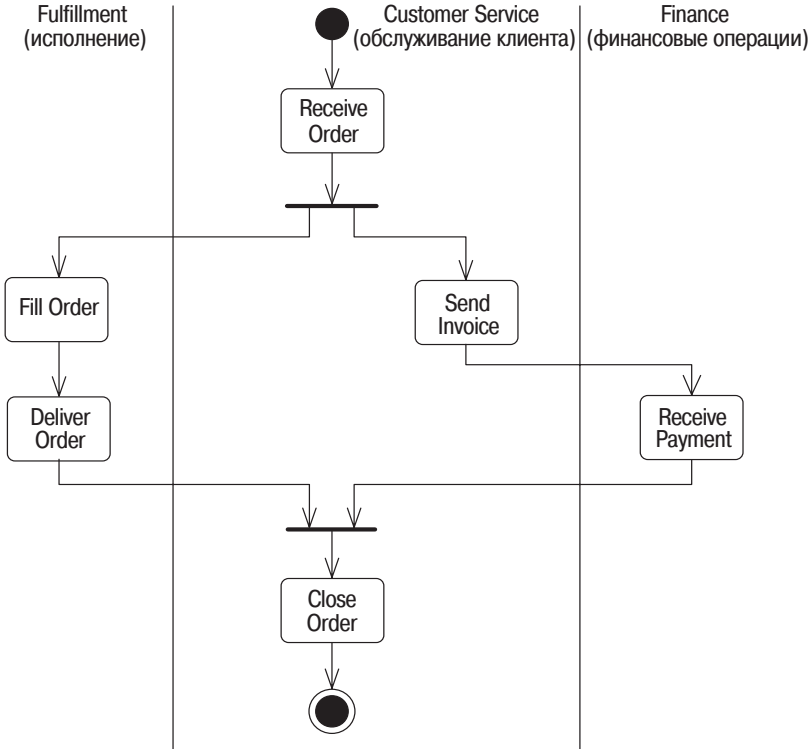

Рис. 11.4. *Разбиение диаграммы деятельности на разделы*

На рис. 11.4 представлено простое одномерное разбиение. Этот способ по понятным причинам часто называют **плавательными дорожками** (swim lanes), и такая форма была единственной в UML 1. В UML 2 сетка может быть двумерной, поэтому «плавательная» метафора больше не содержит воды. Кроме того, можно взять каждое измерение и разделить строчки на столбцы, создавая тем самым иерархическую структуру.

Сигналы

В простом примере на рис. 11.1 диаграммы деятельности имеют четко определенную стартовую точку, соответствующую вызову программы или процедуры. Кроме того, операции могут отвечать на сигналы.

Временной сигнал (time signal) приходит по прошествии времени. Такие сигналы могут означать конец месяца в отчетном периоде или приходить каждую секунду в контроллере реального времени.

На рис. 11.5 показана деятельность, в которой ожидаются два сигнала. Сигнал показывает, что данная деятельность принимает сообщение о событии от внешнего процесса. Это означает, что деятельность постоянно прослушивает эти сигналы, а диаграмма определяет, как деятельность на них реагирует.

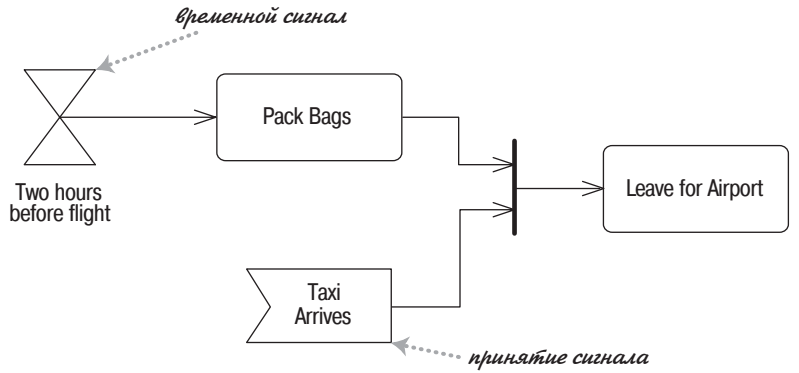

***Рис. 11.5.** Сигналы в диаграмме деятельности*

В случае, показанном на рис. 11.5, до моего отлета остается два часа (`Two hours before flight`), и мне пора собирать багаж. Если я упакую его раньше времени, то все равно не смогу уехать, пока не прибудет такси. Если такси приходит (`Taxi Arrives`) до того, как я успею собрать багаж (`Pack Bags`), то оно должно ждать меня, пока я не закончу.

Мы можем как принимать сигналы, так и посылать их. Это полезно, когда мы посылаем сообщение, а затем должны ожидать ответа, перед тем как продолжить. На рис. 11.6 показан хороший пример этого процесса, основанный на общей идиоме таймаутов. Заметим, что в этой

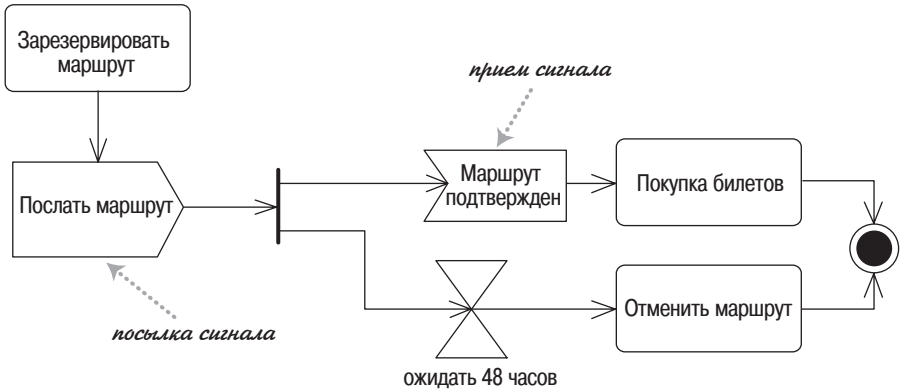

***Рис. 11.6.** Отправка и прием сигналов*

гонке участвует два потока: первый, достигший финального состояния, выигрывает и прерывает выполнение другого потока.

Хотя блоки приема сигналов только ожидают внешнего события, мы можем также показать входящий в него поток. Это означает, что мы не начинаем прослушивание до тех пор, пока поток не инициирует прием.

Маркеры

Смельчаки, рискнувшие погрузиться в дьявольскую глубину спецификаций UML, обнаружат, что в разделе, посвященном активности, много говорится о маркерах (tokens), их создании и использовании. Начальный узел создает маркер, который затем передается следующей процедуре, которая выполняется и передает маркер следующей процедуре. В точку ветвления приходит один маркер, а ветвление порождает маркер для каждого исходящего потока. Наоборот, в точке объединения при появлении отдельного маркера ничего не происходит до тех пор, пока не соберутся все маркеры, затем порождается маркер для исходящего потока.

Можно визуализировать маркеры с помощью монеток или счетчиков, перемещающихся по диаграмме. В случае более сложных диаграмм деятельности маркеры часто облегчают визуализацию.

Потоки и ребра

В UML 2 параллельно употребляются термины **поток** (flow) и **ребро** (edge) для обозначения связи между двумя операциями. Самый простой вид ребра – это обычная стрелка между двумя операциями. Если хотите, можете присвоить ей имя, но в большинстве случаев простой стрелки будет достаточно.

При возникновении трудностей с разводкой линий можно воспользоваться разъемами (connectors), которые позволят вам не рисовать линии на всем их протяжении. Разъемы изображаются парами: один для входного и один для выходного потоков, при этом они должны иметь одну и ту же метку. Я предпочитаю без необходимости не применять разъемы, поскольку они нарушают визуализацию потока управления.

Простейшие ребра передают маркер, имеющий значение только для управления потоком. Однако по ребрам можно передавать объекты; тогда объекты будут играть роль маркеров как передатчиков данных. Передачу объекта вдоль ребра можно показать, помещая на ребро прямоугольник класса. Можно также изображать контакты (pins) на операциях, хотя использование контактов имеет некоторые хитрости, о которых я кратко расскажу.

Все способы, показанные на рис. 11.7, эквивалентны. Вы вправе выбрать тот способ, который лучше всего отражает то, что вы хотите сообщить. В большинстве случаев вполне достаточно простой стрелки.

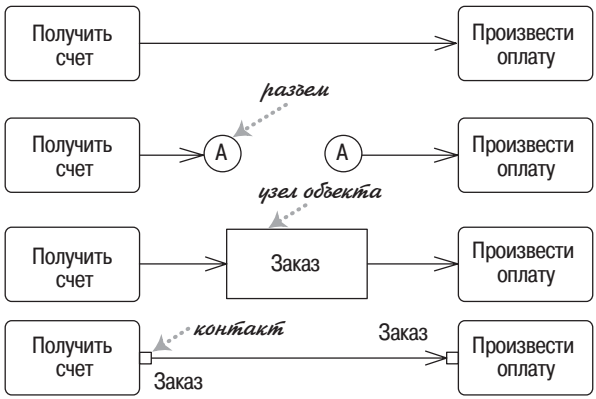

Рис. 11.7. *Четыре способа представления ребер*

Контакты и преобразования

Процедуры, как и методы, могут иметь параметры. Показывать на диаграмме деятельности информацию о параметрах не обязательно, но при желании можно отобразить параметры с помощью **контактов** (pins). Если процедура разбивается на части, то контакты должны соответствовать прямоугольникам параметров на разделенной диаграмме.

Если требуется нарисовать точную диаграмму деятельности, то необходимо обеспечить соответствие выходных параметров одной процедуры входным параметрам другой. Если они не совпадают, то можно указать **преобразование** (transformation) (рис. 11.8) для перехода от одной процедуры к другой. Преобразование должно представлять собой

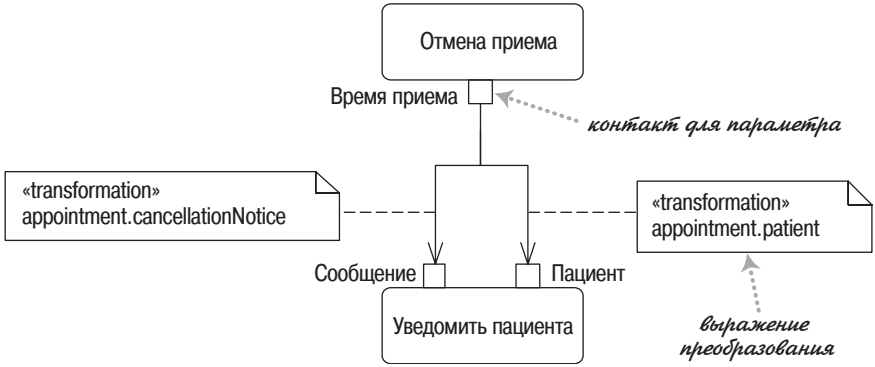

***Рис. 11.8.** Преобразование потока*

выражение, свободное от сторонних эффектов: главное, чтобы запрос с выходного контакта предоставлял тип объекта, соответствующий входному контакту.

Показывать контакты на диаграмме деятельности не обязательно. Контакты удобны, когда требуется увидеть данные, принимаемые и передаваемые различными процедурами. При моделировании бизнес-процессов посредством контактов можно отображать ресурсы, которые потребляются и производятся различными процедурами.

С помощью контактов можно без опаски показать несколько потоков, входящих в одну и ту же операцию. Нотация контактов усиливает предположение о наличии последующего объединения, а в UML 1 вообще нет контактов, поэтому не возникает путаницы с более ранними допущениями.

Области расширения

При работе с диаграммами деятельности часто сталкиваешься с ситуациями, когда выход одной операции инициирует многочисленные вызовы другой операции. Есть несколько способов показать это, но лучше всего подходит область расширения. **Область расширения** (expansion region) отмечает область диаграммы деятельности, где операции выполняются один раз для каждого элемента коллекции.

На рис. 11.9 процедура `Choose Topics` (Выбрать темы) генерирует список тем. Затем каждый элемент этого списка становится маркером для входа процедуры `Write Article` (Написать статью). Подобным образом каждая операция `Review Article` (Рецензировать статью) генерирует единственную статью, которая добавляется к выходному списку области расширения. Когда все маркеры в области расширения достигают выходной коллекции, область расширения генерирует единственный маркер для списка, который передается процедуре `Publish Newsletter` (Опубликовать информационный бюллетень).

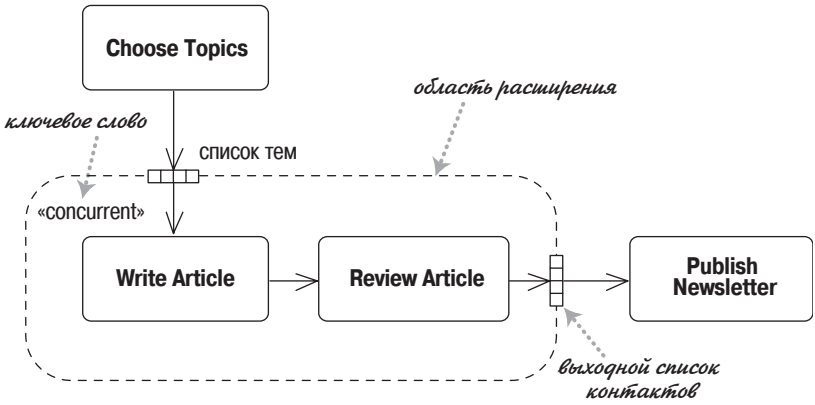

Рис. 11.9. *Область расширения*

В данном случае в выходной и входной коллекциях одинаковое количество элементов. Однако в выходной коллекции может оказаться меньше элементов, чем во входной; в таком случае область расширения действует как фильтр.

На рис. 11.9 все статьи пишутся и рецензируются параллельно, что отмечено ключевым словом «concurrent». Область расширения также может быть итеративной. Итеративные области должны полностью обрабатывать каждый входной элемент за один раз.

Если есть только одна операция, которую надо вызывать несколько раз, то применяется нотация, показанная на рис. 11.10. Такой способ записи предполагает параллельное расширение, поскольку оно наиболее общее. Эта нотация соответствует концепции динамического параллелизма, принятой в UML 1.

Рис. 11.10. *Нотация для единственной процедуры в области расширения*

Окончание потока

При получении нескольких маркеров, как в случае с областью расширения, поток часто останавливается, даже если вся активность в целом не завершена. Окончание потока (flow final) означает завершение конкретного потока без завершения всей активности.

На рис. 11.11 показана модифицированная версия рис. 11.9, в которой статьи могут быть отвергнуты. Если статья отклонена, то маркер ликвидируется окончанием потока. В отличие от окончания активно-

сти, в данном случае остальная активность может продолжаться. Этот подход позволяет областям расширения действовать в качестве фильтров, в результате чего выходная коллекция будет меньше входной.

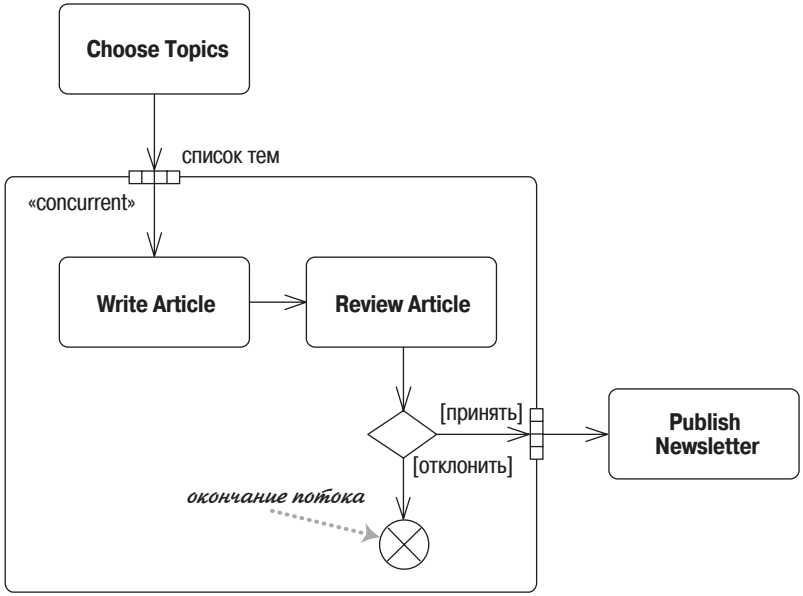

Рис. 11.11. *Окончание потока в активности*

Описания объединений

По умолчанию объединение разрешает выполнение выходного потока, когда все входные потоки достигли объединения. (Или, говоря более формальным языком, оно порождает маркер выходного потока, когда приходят маркеры всех входных потоков.) В некоторых случаях, в частности, когда есть поток с несколькими маркерами, полезно иметь более сложное правило.

Описание объединения (join specification) – это логическое выражение, присоединенное к объединению. Каждый раз, когда в объединение прибывает маркер, вычисляется описание объединения, и если его значение истинное, то порождается маркер. Поэтому на рис. 11.12, независимо от того, выбираю ли я напиток (`Select Drink`) или кидаю монетку (`Insert Coin`), автомат оценивает определение объединения. Автомат утоляет мою жажду, только если я кинул достаточное количество денег. Если, как в данном случае, вы хотите показать, что вы приняли маркер в каждом входном потоке, то необходимо именовать потоки и включить их в описание объединения.

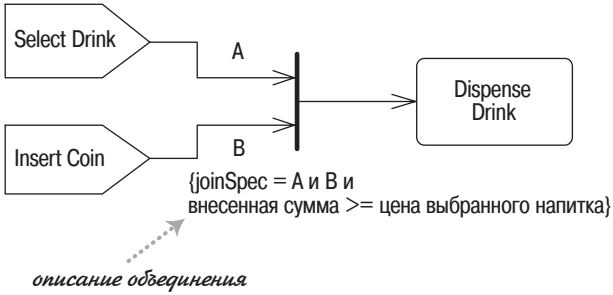

Рис. 11.12. *Описание объединения*

И еще немного

Я должен подчеркнуть, что эта глава лишь слегка затронула диаграммы деятельности. Учитывая объем языка UML, можно написать целую книгу только об одной этой технологии. На самом деле я думаю, что диаграммы деятельности могли бы стать отличной темой для книги, посвященной пристальному рассмотрению нотаций и их применению.

Жизненно важный вопрос заключается в том, как широко они применяются. В настоящий момент диаграммы деятельности не относятся к наиболее распространенным технологиям языка UML, и все их предшественники в области моделирования потоков также не пользовались успехом. Технология диаграмм еще не достигла необходимого уровня для описания поведения таким способом. С другой стороны, в многочисленных сообществах есть проявления скрытой потребности, которую могли бы удовлетворить стандартные приемы.

Когда применяются диаграммы деятельности

Самое большое достоинство диаграмм деятельности заключается в том, что они поддерживают и стимулируют применение параллельных процессов. Именно благодаря этому они представляют собой мощное средство моделирования потоков работ. Множество импульсов к развитию UML 2 пришло от людей, вовлеченных в эти потоки работ.

Можно также применять диаграмму деятельности в качестве совместимой с языком UML блок-схемы. Хотя это позволяет разрабатывать блок-схемы, близкие к UML, но вряд ли это очень захватывающий процесс. В принципе, можно воспользоваться преимуществами, предоставляемыми ветвлением и объединением, для описания параллельных алгоритмов одновременно выполняющихся программ. Хотя сам я не очень активно применял параллельные циклы, но у меня также нет достаточного количества подтверждений этого от людей, имеющих большой опыт их применения. Я думаю, причина в том, что сложность

параллельного программирования состоит в противостоянии данных параллельных процессов, а диаграммы деятельности не могут оказать большой помощи в этом вопросе.

В наибольшей степени их мощь может проявиться в случае применения UML как языка программирования. Здесь диаграммы деятельности являются ценным инструментом для представления логики поведения систем.

Мне часто приходилось видеть, как диаграммы деятельности применялись для описания прецедентов. Опасность такого подхода в том, что часто эксперты в предметной области с трудом могут им следовать. Если дело обстоит так, то лучше обойтись обычной текстовой формой.

Где найти дополнительную информацию

Хотя диаграммы деятельности всегда были довольно сложным инструментом, а в UML 2 они стали еще сложнее, тем не менее не существует хорошей книги, в которой бы они описывались достаточно глубоко. Я надеюсь, что этот пробел когда-нибудь будет восполнен.

Различные ориентированные на потоки технологии по своему стилю напоминают диаграммы деятельности. Одна из лучших (но едва ли широко известная) – это Petri Nets, информацию о которой можно найти на веб-сайте *http://www.daimi.au.dk/PetriNets/*.

12

Коммуникационные диаграммы

Коммуникационные диаграммы (communication diagrams) – это особый вид диаграмм взаимодействия, акцентированных на обмене данными между различными участниками взаимодействия. Вместо того чтобы рисовать каждого участника в виде линии жизни и показывать последовательность сообщений, располагая их по вертикали, как это делается в диаграммах последовательности, коммуникационные диаграммы допускают произвольное размещение участников, позволяя рисовать связи, показывающие отношения участников, и использовать нумерацию для представления последовательности сообщений.

В UML 1.x эти диаграммы назывались диаграммами кооперации (collaboration diagrams). Это подходящее название, и я подозреваю, что оно будет существовать, пока люди не привыкнут к новому названию. (Между этим понятием и кооперацией есть различие (*стр. 161*); отсюда изменение названия.)

На рис. 12.1 приведена коммуникационная диаграмма для централизованного управления, показанного на рис. 4.2. С помощью коммуникационной диаграммы можно увидеть, как участники связаны друг с другом.

Кроме отображения связей, которые представляют собой экземпляры ассоциаций, можно также показать временные связи, возникающие только в контексте взаимодействия. В данном случае связь «local» (локальная) от объекта Order (Заказ) к объекту Product (Продукт) – это локальная переменная, а другими временными связями являются «parameter» (параметр) и «global» (глобальная). Эти ключевые слова употреблялись в UML 1, но пропали из UML 2. Они полезны, поэтому я надеюсь, что разработчики от них не откажутся.

Стиль нумерации на рис. 12.1 простой и общеупотребительный, но в языке UML он не разрешен. В соответствии с правилами UML необходимо придерживаться вложенной десятичной нумерации, как показано на рис. 12.2.

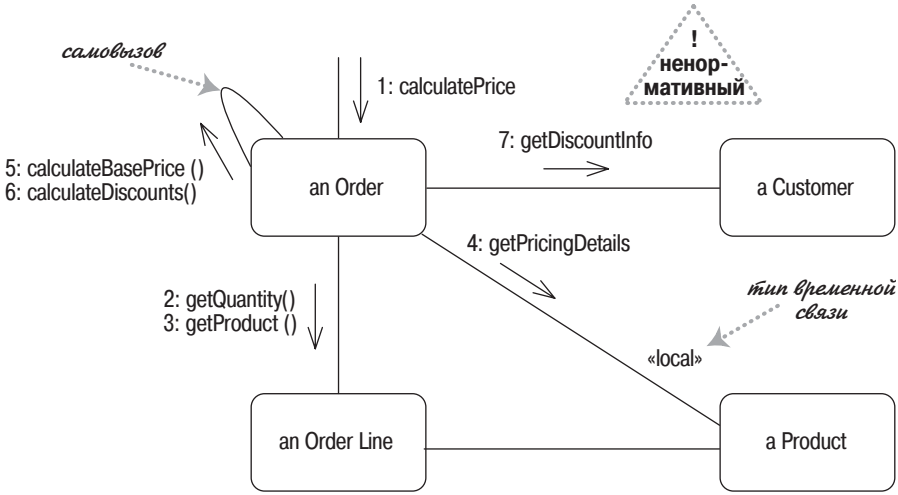

Рис. 12.1. *Коммуникационная диаграмма системы централизованного управления*

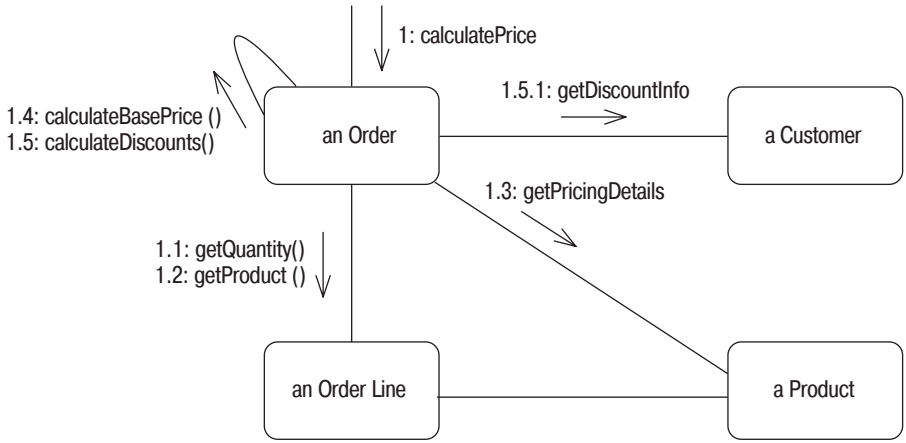

Рис. 12.2. *Коммуникационная диаграмма с вложенной десятичной нумерацией*

Вложенная десятичная нумерация нужна, потому что требуется исключить неопределенность при самовызовах. На рис. 4.2 четко показано, что метод `getDiscountInfo` вызывается из метода `calculateDiscount`. Однако в случае применения линейной нумерации, как на рис. 12.1, нельзя будет сказать, вызывается ли метод `getDiscountInfo` из метода `calculateDiscount` или из более общего метода `calculatePrice`. Схема вложенной нумерации позволяет обойти эту трудность.

Несмотря на ее неправомерность, многие специалисты предпочитают линейную схему нумерации. Вложенная нумерация может быть очень

сложной, в частности потому, что вызовы могут иметь большой уровень вложенности, что приводит к такой последовательности номеров, как 1.1.1.2.1.1. В таких случаях лекарство от неопределенности хуже болезни.

Кроме чисел в сообщениях можно увидеть и буквы. Эти символы обозначают различные потоки управления. Так, A5 и B2 могут быть различными потоками; сообщения 1a1 и 1b1 могут быть различными потоками, параллельно вложенными в сообщение 1. Буквы, обозначающие потоки, можно увидеть также и на диаграммах последовательности, хотя это и не дает визуального представления о параллельности.

Коммуникационные диаграммы не имеют точно определенных нотаций для управляющей логики. Они допускают применение маркеров итерации и защиты (*стр. 86*), но не позволяют полностью определить алгоритм управления. Не существует также специальных обозначений для создания и удаления объектов, но ключевые слова «create» и «delete» соответствуют общепринятым соглашениям.

Когда применяются коммуникационные диаграммы

Основной вопрос, связанный с коммуникационными диаграммами, заключается в том, в каких случаях надо предпочесть их, а не более общие диаграммы последовательности. Ведущую роль в принятии такого решения играют личные предпочтения: у людей разные вкусы. Чаще всего именно это определяет тот или иной выбор. В целом, большинство специалистов, по-видимому, предпочитает диаграммы последовательности, а что касается меня, то я поддерживаю большинство.

Более рациональный подход утверждает, что диаграммы последовательности удобнее, если вы хотите подчеркнуть последовательность вызовов, а коммуникационные диаграммы лучше выбрать, когда надо акцентировать внимание на связях. Многие специалисты считают, что коммуникационные диаграммы проще модифицировать на доске, поэтому они хорошо подходят для рассмотрения вариантов, хотя я в таких случаях обычно предпочитаю CRC-карточки.

13

Составные структуры

Одной из наиболее значимых новых черт языка UML 2 является возможность превращать класс в иерархию внутренних структур. Это позволяет разбить сложный объект на составляющие.

На рис. 13.1 показан класс `TV Viewer` (Телевизор) вместе с интерфейсами, которые он предоставляет и которые требует (*стр. 96*). Я показал его двумя способами: с помощью шарово-гнездовой нотации и с помощью перечисления внутри объекта.

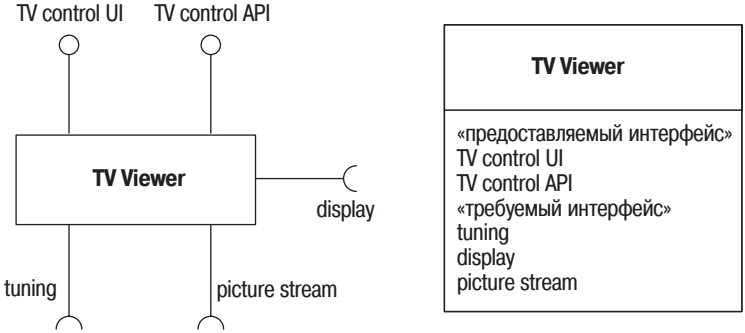

Рис. 13.1. *Два способа представления объекта TV Viewer и его интерфейсов*

На рис. 13.2 показан этот же класс, разбитый внутри на две части, которые предоставляют и требуют различные интерфейсы. Каждая часть имеет имя в виде `имя : класс`, в котором каждый из составляющих его элементов может отсутствовать. Составляющие части не являются описанием экземпляров, поэтому они выделены жирным шрифтом, а не подчеркнуты.

Можно также указать количество экземпляров конкретной части. На рис. 13.2 показано, что каждый объект `TV Viewer` содержит один генератор (`Generator`) и один блок управления (`controls`).

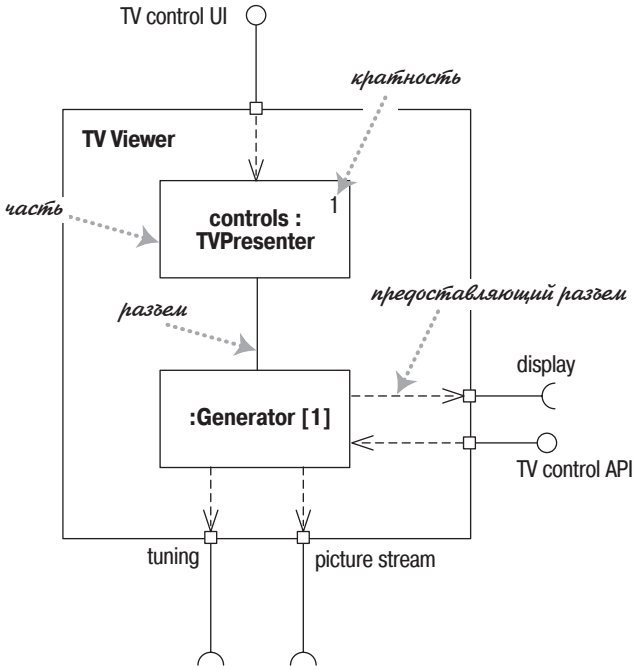

Рис. 13.2. *Внутренний вид компонента (пример, предложенный Джимом Рамбо)*

Чтобы обозначить часть, реализующую интерфейс, надо нарисовать разъем, предоставляемый этим интерфейсом. Аналогично, чтобы показать часть, нуждающуюся в интерфейсе, надо нарисовать разъем, предоставляемый этому интерфейсу. Кроме того, разъемы между частями можно показать или с помощью простой линии, как сделано в данном случае, или с помощью шарово-гнездовой нотации (*стр. 98*).

К внешней структуре можно добавить порты (рис. 13.3). Порты позволяют группировать требуемые и предоставляемые интерфейсы в логи-

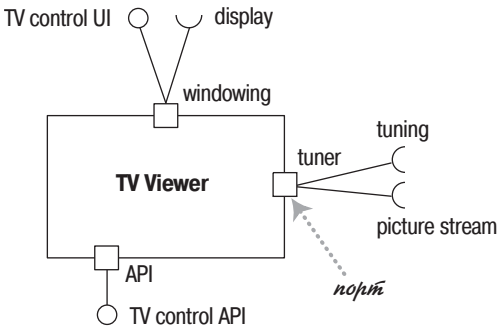

Рис. 13.3. *Компонент с несколькими портами*

ческие взаимодействия, которые компонент имеет с объектами внешнего мира.

Когда применяются составные структуры

Составные структуры являются нововведением в UML 2, хотя некоторые прежние методы реализовывали подобные идеи. Правильный подход к различию между пакетами и составными структурами заключается в том, что пакеты представляют группы времени компиляции, а составные структуры представляют группы времени выполнения. А раз так, то они лучше подходят для показа компонентов и способов их разбиения на части; следовательно, множество этих нотаций применяется в диаграммах компонентов.

Поскольку составные структуры – новый элемент языка UML 2, то слишком рано говорить об эффективности их практического применения; многие члены сообщества UML думают, что эти диаграммы станут весьма ценным дополнением.

14

Диаграммы компонентов

В объектно-ориентированном сообществе идут дебаты о том, в чем состоит различие между компонентом и обычным классом. Я не хочу обсуждать здесь этот спорный вопрос, но могу показать нотацию языка UML, используемую, чтобы отличить их друг от друга.

В UML 1 был отдельный символ для компонента (рис. 14.1). В UML 2 этого значка нет, но можно обозначить прямоугольник класса похожим значком. Или можно воспользоваться ключевым словом «component» (компонент).

Рис. 14.1. *Нотация для компонентов*

Кроме этого значка компоненты не принесли с собой никаких новых обозначений. Компоненты связываются между собой с помощью предоставляемых или требуемых интерфейсов, при этом шарово-гнездовая нотация (*стр. 98*) обычно применяется только на диаграммах классов. Можно также разбивать компоненты на части с помощью диаграмм составных структур.

На рис. 14.2 показан пример простой диаграммы компонентов. В этом примере компонент `Till` (Касса) может взаимодействовать с компонентом `Sales Server` (Сервер продаж) с помощью интерфейса `sales message` (Сообщение о продажах). Поскольку сеть ненадежна, то компонент `Message Queue` (Очередь сообщений) установлен так, чтобы касса могла общаться с сервером, когда сеть работает, и разговаривать с очередью сообщений, когда сеть отключена. Тогда очередь сообщений сможет поговорить с сервером, когда сеть снова станет доступной. В результате

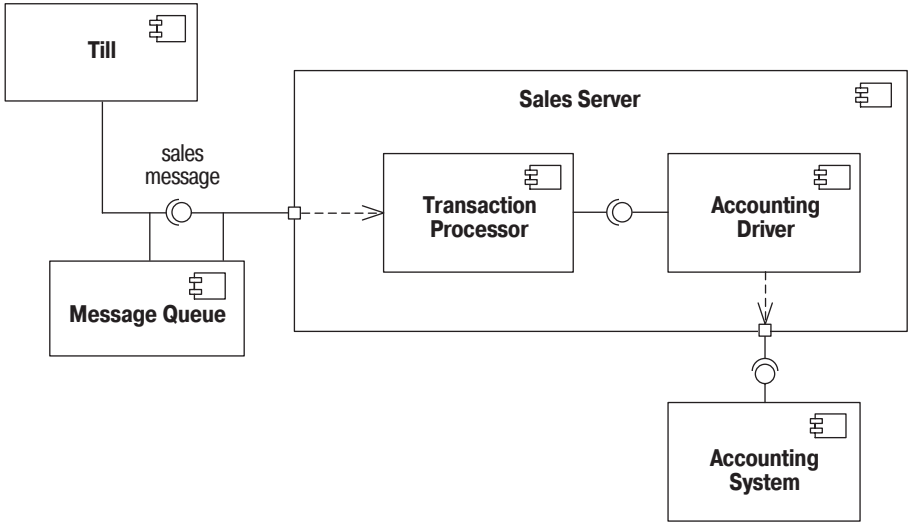

Рис. 14.2. *Пример диаграммы компонентов*

очередь сообщений предоставляет интерфейс для разговора с кассой, и требует такой же интерфейс для разговора с сервером. Сервер разделен на два основных компонента: `Transaction Processor` (Процессор транзакций) реализует интерфейс сообщений, а `Accounting Driver` (Драйвер счетов) общается с `Accounting System` (Система ведения счетов).

Как я уже говорил, вопрос о сущности компонента является предметом бесконечных споров. Вот одно из наиболее продуманных суждений, обнаруженных мною:

> *Компоненты – это не технология. Технические специалисты считают их трудными для понимания. Компоненты – это скорее стиль отношения клиентов к программному обеспечению. Они хотят иметь возможность покупать необходимое им программное обеспечение частями, а также иметь возможность обновлять его, как они обновляют свою стереосистему. Они хотят, чтобы новые компоненты работали так же, как и прежние, и обновлять их согласно своим планам, а не по указанию производителей. Они хотят, чтобы системы различных производителей могли работать вместе и были взаимозаменяемыми. Это очень разумные требования. Одна загвоздка: их трудно выполнить.*
>
> *Ральф Джонсон (Ralph Johnson),*
> *http://www.c2.com/cgi/wiki?DoComponentsExist*

Важно то, что компоненты представляют элементы, которые можно независимо друг от друга купить и обновить. В результате разделение системы на компоненты является в большей мере маркетинговым решением, чем техническим. Прекрасное руководство по данному вопросу представляет книга Хохмана [23]. Она также напоминает о том,

что следует остерегаться разделения системы на слишком мелкие компоненты, поскольку очень большим количеством компонентов трудно управлять, особенно когда производство версий поднимает свою уродливую голову; отсюда пошло выражение «ад DLL». В ранних версиях языка UML компоненты применялись для представления физических структур, таких как DLL. Теперь это не актуально; в настоящее время эта задача решается при помощи артефактов (artifacts) (*стр. 121*).

Когда применяются диаграммы компонентов

Диаграммы компонентов следует применять, когда система разделяется на компоненты и надо показать их взаимоотношения посредством интерфейсов или схему компонентов в низкоуровневой структуре системы.

15

Кооперации

В отличие от других глав этой книги, в данной не обсуждаются формальные диаграммы UML 2. В стандарте UML кооперации рассматриваются как часть составных структур, но диаграмма кооперации в действительности совершенно отличается от составных структур и применялась в UML 1 без всякой связи с составными структурами. Поэтому я счел, что лучше обсудить кооперации в отдельной главе.

Рассмотрим систему обозначений аукциона. В любом аукционе (`Auction`) могут участвовать продавец (`seller`), покупатели (`buyer`), множество вещей (`lot`) и какие-либо предложения о покупке (`offer`). Мы можем описать эти элементы в терминах диаграммы классов (рис. 15.1) и, возможно, посредством нескольких диаграмм взаимодействий (рис. 15.2).

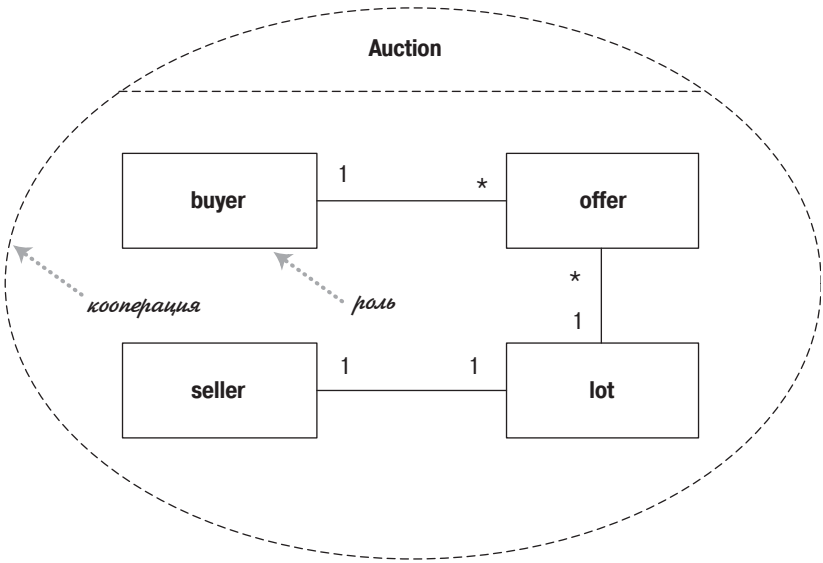

Рис. 15.1. *Кооперация вместе с ее классами и ролями*

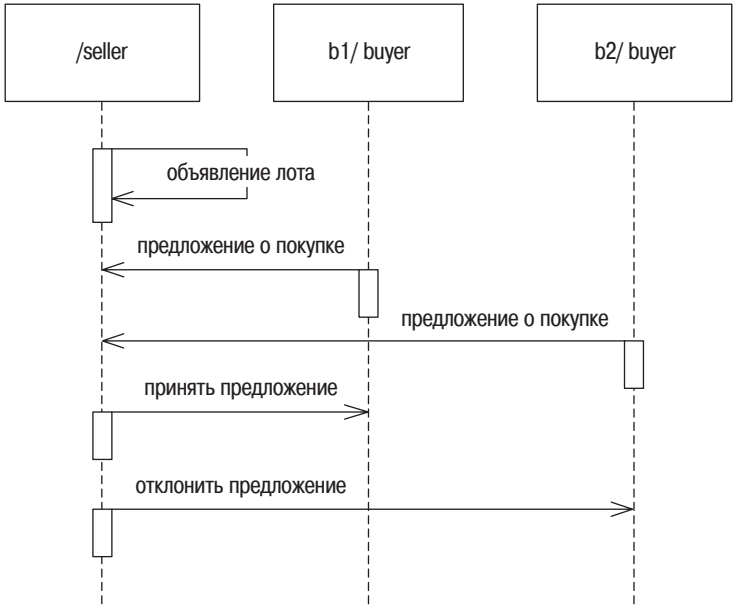

Рис. 15.2. *Диаграмма последовательности для аукционной кооперации*

На рис. 15.1 представлена не совсем обычная диаграмма классов. Во-первых, она обведена пунктирным эллипсом, который представляет аукционную кооперацию. Во-вторых, так называемые классы в кооперации – это не классы, а **роли** (roles), которые реализовываются в процессе выполнения кооперации, поэтому их имена начинаются с маленькой буквы. Сопоставление фактических интерфейсов или классов ролям кооперации не является чем-то необычным, но тем не менее вы не обязаны это делать.

Как видите, на диаграмме взаимодействий участники именуются немного необычно. В кооперации схема именования выглядит следующим образом: `имя-участника / имя-роли : имя-класса`. Как всегда, все эти элементы необязательны.

Применение кооперации можно обозначить, отмечая ее наличие на диаграмме классов, как показано на рис. 15.3, где представлены некоторые классы приложения. Связи, идущие от кооперации к этим классам, показывают, как классы играют различные роли, определенные в кооперации.

В языке UML предполагается, что можно показать применение паттернов, но вряд ли автор каких-либо паттернов будет это делать. Эрих Гамма (Erich Gamma) разработал прекрасную альтернативную нотацию (рис. 15.4). Элементы диаграммы обозначаются либо именем паттерна, либо комбинацией `паттерн:роль`.

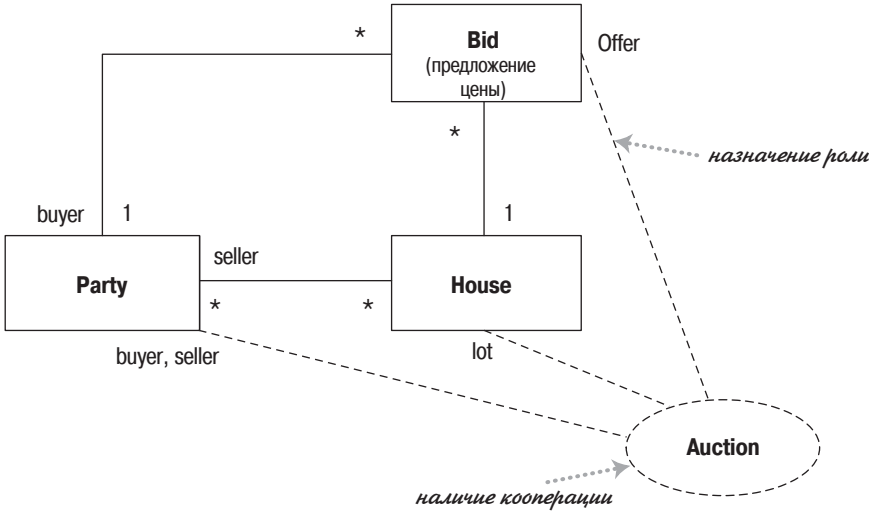

Рис. 15.3. *Наличие кооперации*

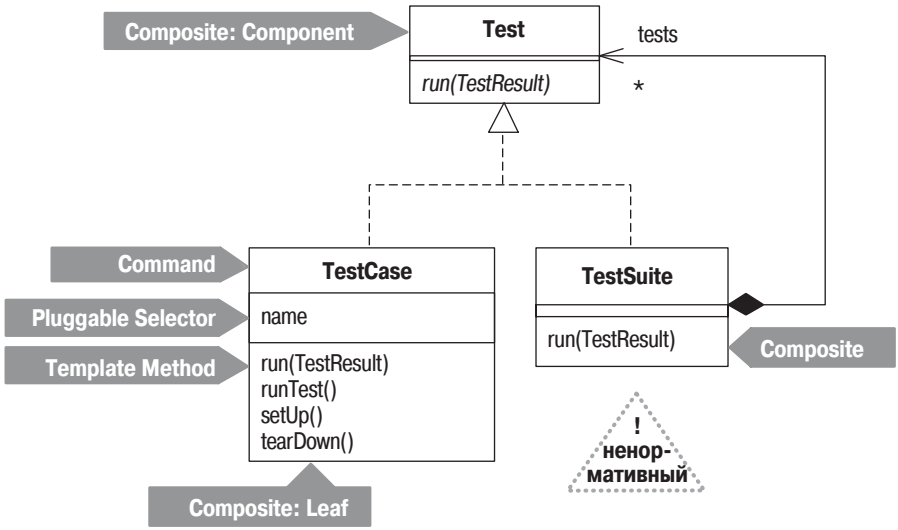

Рис. 15.4. *Необычный способ показа применения паттерна в JUnit (junit.org)*

Когда применяются кооперации

Кооперации существуют со времен UML 1, но я признаю, что вряд ли применял их даже для разработки паттернов. Кооперации предоставляют способ группирования элементов взаимодействия, когда роли исполняются различными классами. Однако на практике я не встречал, чтобы этот тип диаграмм кого-то покорил.

16

Диаграммы обзора взаимодействия

Диаграммы обзора взаимодействия – это комбинация диаграмм деятельности и диаграмм последовательности. Можно считать диаграммы обзора взаимодействия диаграммами деятельности, в которых деятельности заменены небольшими диаграммами последовательности, или диаграммами последовательности, разбитыми с помощью нотации диаграмм деятельности для отображения потока управления. В любом случае они представляют довольно необычную смесь.

На рис. 16.1 показан пример простой диаграммы такого типа; нотация нам уже знакома по главам, посвященным диаграммам деятельности и диаграммам последовательности. В этой диаграмме мы хотим составить и отформатировать отчетный доклад о заказах. Если клиент внешний, то информацию поставляет XML, а если внутренний, то информация берется из базы данных. Небольшие диаграммы последовательности показывают две альтернативы. После получения данных мы форматируем отчет; в этом случае мы не представляем диаграмму последовательности, а просто ссылаемся на нее.

Когда применяются диаграммы обзора взаимодействия

Этот тип диаграмм появился в UML 2, поэтому еще трудно понять, насколько успешно они решают практические задачи. Я от них не в восторге, поскольку они сочетают два разных стиля, и я считаю, что сочетание это не очень удачное. На какой диаграмме остановиться – на диаграмме деятельности или на диаграмме последовательности – нужно решать на основании того, какая из них вам лучше подходит.

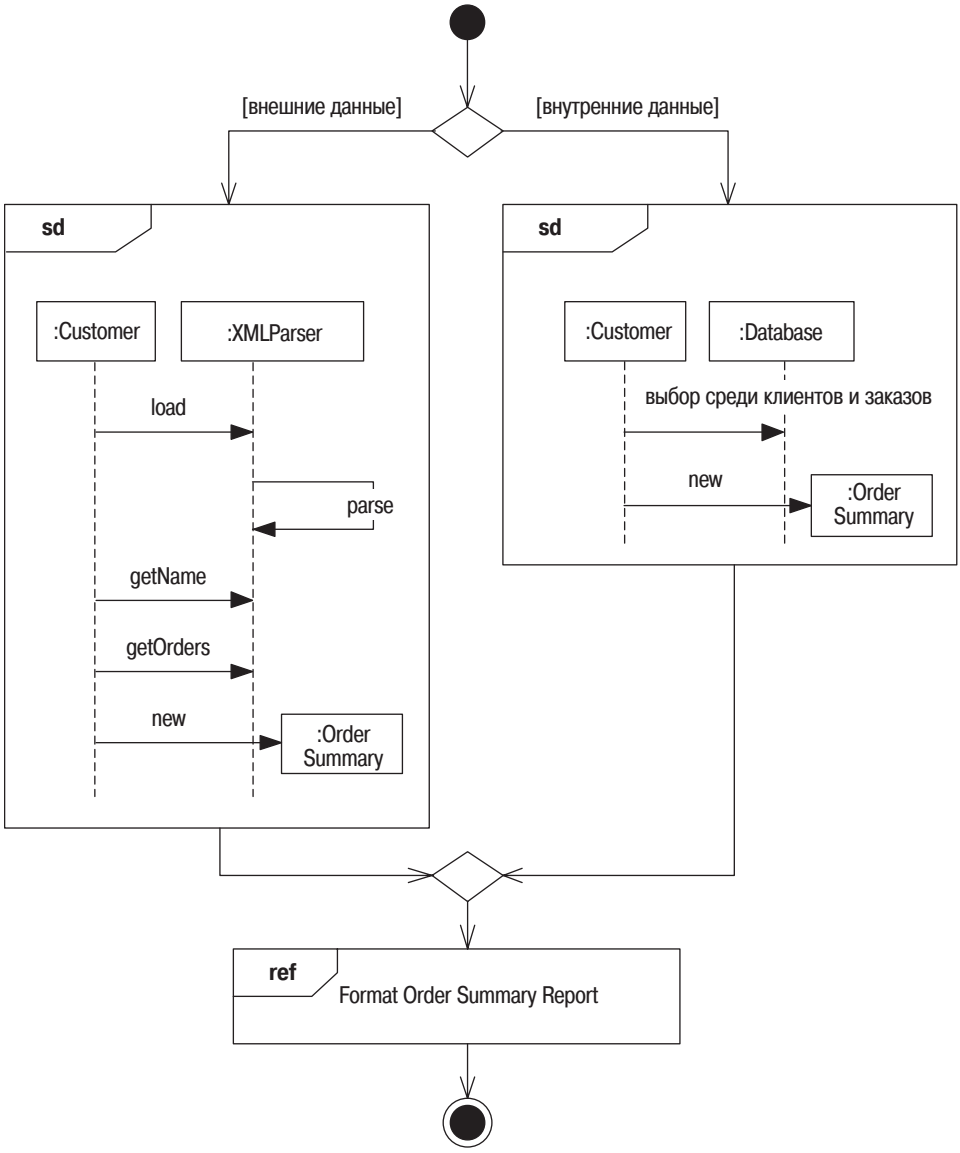

Рис. 16.1. *Диаграмма обзора взаимодействий*

17

Временные диаграммы

После окончания средней школы и до того как начать свою компьютерную карьеру, я работал инженером-электронщиком. Поэтому я испытываю щемящее чувство узнавания, когда вижу, как язык UML определяет временные диаграммы в качестве своих стандартных диаграмм. Временные диаграммы существовали в электронной промышленности испокон веков, и никто не мог подумать, что потребуется помощь UML, чтобы понять их назначение. Но уж раз они появились в UML, то заслужили краткого упоминания.

Временные диаграммы – это еще одна форма диаграмм взаимодействия, которая акцентирована на временных ограничениях: либо для одиночного объекта, либо, что более полезно, для группы объектов. Давайте рассмотрим простой сценарий, основанный на использовании насоса (`Pump`) и нагревательного элемента (`Hotplate`) в кофеварке (coffee pot). Представим себе правило, которое гласит, что между включением насоса и включением нагревательного элемента должно пройти по крайней мере 10 секунд. Когда емкость с водой становится пустой (`waterEmpty`), насос выключается, а нагревательный элемент не может оставаться включенным более 15 минут.

На рис. 17.1 и 17.2 показаны альтернативные способы представления таких временных ограничений. Главное различие состоит в том, что на рис. 17.1 изменения состояния обозначаются переходом от одной горизонтальной линии к другой, а на рис. 17.2 горизонтальное расположение остается таким же, а изменения состояния обозначаются перекрещиванием горизонтальных линий. Стиль, представленный на рис. 17.1, следует предпочесть, когда состояний немного, а стиль, показанный на рис. 17.2, лучше подходит, когда имеешь дело с большим количеством состояний.

Пунктирные линии, при помощи которых я обозначил временные границы `{>10s}`, не обязательны. Если вы считаете, что они помогут точно определить, какие события вызывают временные ограничения, то нарисуйте их.

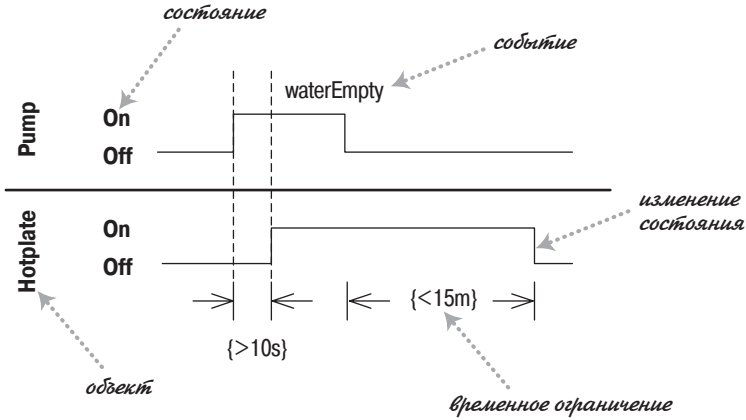

Рис. 17.1. *Временная диаграмма, на которой состояния представлены в виде линий*

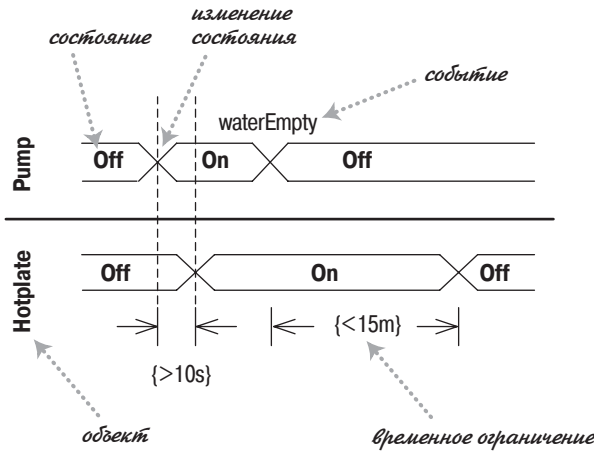

Рис. 17.2. *Временная диаграмма, на которой состояния представлены в виде областей*

Когда применяются временные диаграммы

Временные диаграммы полезны для обозначения временных интервалов между изменениями состояний различных объектов. Кроме того, эти диаграммы знакомы инженерам по оборудованию.

A

Отличия версий языка UML

Когда вышло в свет первое издание этой книги, язык UML имел еще версию 1.0. С ее появлением термины для многих элементов языка UML устоялись, а консорциум OMG получил официальное признание. С тех пор версии языка UML пересматривались несколько раз. В этом приложении описываются все существенные изменения языка UML с момента появления версии 1.0 и то, как эти изменения повлияли на материал данной книги. Эволюция языка UML потребовала обновления книги, и третье издание содержит материал, отражающий ситуацию на момент 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 метод стал называться UML.

В январе 1997 года компания Rational вместе с группой партнеров представила на рассмотрение инициативной группы анализа и проектирования из OMG версию 1.0 языка UML. В дальнейшем компания Rational и другие участники объединили свои усилия и в сентябре 1997 года предложили в качестве 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.

Затем последовала целая серия переработок языка UML. Версия 1.1 появилась в 1998 году, версия 1.3 в 1999, 1.4 в 2001 и 1.5 в 2002. Большинство изменений в версиях 1.x в основном были скрыты в глубине UML, за исключением версии 1.3, изменения в которой были явно видны, особенно это касается прецедентов и диаграмм деятельности.

Хотя выпуск версий UML 1 продолжался, разработчики UML основное внимание стали уделять UML 2. Первый RFP (Request for Proposals – запрос на предложение) был объявлен в 2000, но UML 2 не был достаточно стабилен вплоть до 2003 года.

Разработка UML почти наверняка будет продолжаться и впредь. Дополнительные сведения можно почерпнуть на форуме по языку UML (*http://uml-forum.com*). Кроме того, некоторую информацию можно найти на моем сайте (*http://martinfowler.com*).

Изменения в первом издании книги

В процессе эволюции языка UML я пытался учесть его изменения, что привело к появлению нескольких вариантов книги «UML. Основы». При этом я пользовался хорошей возможностью исправить ошибки и сделать изложение более ясным.

Наиболее динамичным периодом внесения изменений было время печати первого издания, когда нам часто приходилось обновлять книгу от выпуска к выпуску, чтобы сохранять соответствие с появляющимися стандартами UML. Выпуски с первого по пятый были основаны на версии UML 1.0. Изменения между этими выпусками были минимальными. В шестой выпуск вошла версия 1.1.

Выпуски с седьмого по десятый в основе имели версию UML 1.2; версия UML 1.3 впервые была использована в одиннадцатом выпуске. На обложках выпусков, базировавшихся на версиях UML после 1.0, ставился номер выпуска.

С первого по шестой выпуски второго издания были основаны на версии UML 1.3. Небольшие изменения, появившиеся в версии UML 1.4, впервые были учтены в седьмом выпуске.

Целью третьего издания было обновление книги до UML 2 (см. табл. A.1). Далее в этом приложении внимание сосредоточено на основных отличиях в языке UML, имевших место при изменениях версий с 1.0 на 1.1, с 1.2 на 1.3. и с 1.x на 2.0. Мне бы не хотелось подробно обсуждать все изменения, поэтому остановлюсь лишь на тех из них, которые каким-то образом затрагивают материал книги «UML. Основы» или относятся к важным свойствам языка UML, рассмотренным в первом издании.

Продолжая придерживаться стиля первого издания, я рассмотрю основные элементы языка UML и особенности применения языка UML при разработке реальных проектов. Как и ранее, выбор материала и соответствующие рекомендации основаны на моем собственном опы-

те. 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 мне о них.

Таблица А.1. Книга «UML. Основы» и соответствующие версии языка

«UML. Основы»	Версии UML
1-е издание	UML 1.0–1.3
2-е издание	UML 1.3–1.4
3-е издание	UML 2.0 и далее

Отличия версий языка UML 1.0 и 1.1

Тип и класс реализации

В первом издании книги «UML. Основы» я рассмотрел различные точки зрения на разработку и те возможные изменения, которые произойдут в результате совершенствования способов изображения и интерпретации моделей, в частности это касается 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 (implementation class) соответствует некоторому классу в контексте разрабатываемой программы. **Тип** (type) 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 подход с точки зрения реализации предполагает использование классов реализации, а концептуальная точка зрения и точка зрения спецификации предполагают использование типов.

Если некоторый класс реализации представляет 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 Java или COM. В этом случае интерфейсы имеют только операции и не имеют атрибутов.

Для класса реализации может быть использована только однозначная статическая классификация, однако в случае типов классификация может быть множественной и динамической. (Я думаю, дело тут в том, что основные объектно-ориентированные языки поддерживают единичную статическую классификацию. Если в один прекрасный день вы будете пользоваться языком, который поддерживает множественную или динамическую классификацию, то это ограничение может быть снято.)

Полные и неполные ограничения классификатора

В предыдущих выпусках «UML. Основы» было отмечено, что ограничение {complete} (полный) 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 {complete}, которое указывает лишь на то, что соответствующее разбиение отражает все подтипы. А это совсем не то же самое. Я обнаружил множество несоответствий в интерпретации этого ограничения, поэтому вам следует обратить на это внимание. Если вы хотите показать, что все экземпляры супертипа должны быть экземпляром одного из подтипов, то во избежание недоразумений я советую использовать другое ограничение. В настоящее время я применяю ограничение {mandatory} (обязательный).

Композиция

Применение композиции в языке UML версии 1.0 означало, что эта связь неизменна (immutable) или заморожена (frozen), по крайней мере, для компонентов с одним значением. Это ограничение больше не является частью определения композиции.

Неизменность и замороженность

Язык UML определяет ограничение {frozen} (замороженный) 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 неизменности я употребляю термин frozen, тем самым я могу применять это ограничение к ролям ассоциаций, классам и атрибутам.

Возвраты на диаграммах последовательности

В UML 1.0 обратное сообщение (или возврат) на диаграмме последовательности вместо сплошной стрелки обозначалось обычной стрелкой (см. предыдущее издание). Это привело к некоторым трудностям, поскольку данное различие трудно заметить, и оно легко приводит к недоразумениям. В UML 1.1 возвраты изображаются пунктирной линией со стрелкой, что мне нравится больше, поскольку делает возвраты намного более очевидными. (Именно поэтому в своей книге «Паттерны анали-

за» [16] я использовал пунктирные возвраты, что представляется мне весьма важным.) Для последующего применения возвратов можно назначить им имена вида `enoughStock :=check()`.

Использование термина «Role»

В языке UML версии 1.0 термин **роль** (role) в основном указывал направление некоторой ассоциации (см. предыдущее издание). В языке UML версии 1.1 данное определение рассматривается как **роль ассоциации** (association role). Помимо нее существует **роль кооперации** (collaboration role), то есть роль, исполняемая некоторым экземпляром класса в кооперации. Версия UML 1.1 придает кооперации еще большую выразительность, и похоже, что такое толкование понятия «роль» может стать ведущим.

Отличия версий языка UML 1.2 (и 1.1) и 1.3 (и 1.5)

Прецеденты

В прецедентах появились новые отношения. В UML 1.1 имелись только два типа отношений между прецедентами: «`uses`» (использует) и «`extends`» (расширяет), каждое их которых является стереотипом обобщения. UML 1.3 предлагает три типа отношений:

- Конструкция «`includes`» (включает) является стереотипом зависимости. Она означает, что выполнение одного прецедента включает в себя другой прецедент. Обычно это отношение встречается в ситуации, когда несколько прецедентов имеют общие этапы или части. Включаемый прецедент может предоставлять другим некоторое общее поведение. В качестве примера можно рассмотреть банкомат (ATM), в контексте которого оба прецедента `Withdraw Money` (Получить деньги) и `Make Transfer` (Сделать перевод) используют прецедент `Validate Customer` (Проверить подлинность клиента). Это отношение в общем случае заменяет применение стереотипа «`uses`».
- **Обобщение** (generalization) прецедента означает, что один прецедент представляет собой вариацию другого. Таким образом, можно иметь один прецедент для сценария «Получить деньги» (базовый вариант использования) и другой прецедент для ситуации, когда выдача денег невозможна по причине отсутствия средств на счету клиента. Отказ от выплаты денег можно представить в виде отдельного прецедента, который уточняет базовый прецедент. (Кроме того, можно определить еще и дополнительный сценарий для прецедента «Получить деньги».) В этом случае специальный прецедент, подобно рассмотренному выше, может изменить какой-либо аспект базового прецедента.
- Конструкция «`extends`» (расширяет) является стереотипом зависимости. Она обеспечивает более управляемую форму расширения по

сравнению с отношением обобщения. В этом случае в базовом прецеденте задается несколько точек расширения. Включающий прецедент может вносить изменения в свое поведение только в этих точках расширения. К примеру, при рассмотрении покупки товара через Интернет можно определить один прецедент для покупки товара с точками расширения для ввода информации о доставке товара и ввода информации об оплате товара. После чего этот прецедент 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 «uses» в ситуациях, когда версия 1.3 рекомендует указывать стереотип «includes», поскольку для многих из них стереотип «includes» может быть заменен стереотипом «uses». И большинство разработчиков применяют стереотип «extends» из версии 1.1 в более широком смысле, предполагая не только отношение «extends» из версии 1.3, но также и важнейшую составляющую отношения обобщения в версии 1.3. Поэтому можно считать, что отношение со стереотипом «extends» расщепляется в версии 1.3 на два отношения: со стереотипом «extends» и обобщение (generalization).

Хотя это объяснение охватывает большую часть известных мне приложений языка UML, в настоящее время мне неизвестен строгий и правильный способ применения в них старых отношений. Однако большинство 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 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.

При разработке условного поведения теперь можно обозначать принятие решения в форме ромба – как для слияния (merge), так и для ветвления (branch). Хотя для описания условного поведения ни ветвления, ни слияния не являются необходимыми, все более общепринятым становится способ изображения, заключающий условное поведение в скобки.

Символ синхронизации в форме черты теперь относится как к **ветвлению** (когда управление расщепляется), так и к **объединению** (когда синхронизируемое управление объединяется снова). Однако теперь никаких дополнительных условий на объединение не накладывается. Необходимо лишь придерживаться правил, гарантирующих соответствие ветвлений и объединений. По существу это означает, что каждое ветвление должно иметь соответствующее объединение, которое соединяет все параллельные нити процесса, берущие начало в исходном ветвле-

нии. Хотя ветвления и объединения могут быть вложенными, их можно удалить с диаграммы, если потоки соединяют ветвления (или объединения) напрямую.

Объединения вступают в силу только тогда, когда все входящие в них потоки завершены. Однако можно определить некоторое условие для исходящего из ветвления потока. Если это условие не выполняется, то соответствующий поток считается завершенным и может участвовать в объединении остальных потоков.

Свойство множественной инициализации больше не поддерживается. Вместо него можно определить динамическую параллельность в некоторой деятельности 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 сответствующему условию синхронизации, хотя такой способ и менее гибок.

Эти правила в какой-то степени уменьшают гибкость диаграмм деятельности, однако они гарантируют, что диаграммы деятельности являются на самом деле частными случаями конечных автоматов. Отношение между диаграммами деятельности и конечными автоматами стало предметом дискуссии инициативной группы RTF. Последующие версии языка UML (после 1.4) вполне могут определить диаграммы деятельности как диаграммы совершенно другой формы.

Отличия версий языка UML 1.3 и 1.4

Наиболее значимым отличием версии 1.4 являются **профили** (profiles), которые позволяют группировать расширения в единое, логически связанное множество. В документацию по языку UML включена пара примеров профилей. Вместе с тем, определение стереотипов стало более формальным, а элементы модели теперь могут иметь несколько стереотипов; в версии UML 1.3 они были ограничены одним стереотипом.

В язык UML были добавлены **артефакты** (artifacts). Артефакт – это физическое олицетворение компонента, так, например, Xerces – это компонент, а все копии файла Xerces.jar на моем жестком диске – это артефакты, которые реализуют компонент Xerces.

До версии 1.3 в метамодели UML не было инструмента для работы с **областью видимости пакетов** (package visibility). Теперь в вашем распоряжении символ «~».

Кроме того, в версии UML 1.4 обычная стрелка обозначает асинхронность на диаграммах взаимодействия – скорее всего, для обратной совместимости. Это затрагивает немногих специалистов, включая меня.

Отличия версий языка UML 1.4. и 1.5

Принципиальным отличием стало введение в язык UML семантики операций – необходимый шаг для превращения UML в язык программирования. Это было сделано, чтобы позволить специалистам работать, не дожидаясь окончания разработки полной версии UML 2.

От UML 1.x к UML 2.0

UML 2 представляет наибольшие изменения, произошедшие в языке. В этой версии изменилось все, и многие изменения коснулись книги «UML. Основы».

Глубокие изменения произошли в метамодели языка UML. И хотя эти изменения не повлияли на книгу, они очень важны для определенных групп специалистов.

Одним из наиболее очевидных изменений стало введение новых типов диаграмм. Диаграммы объектов и диаграммы пакетов широко использовались и прежде, но не были официальными типами диаграмм; теперь это так. В UML 2 диаграммы кооперации теперь называются коммуникационными диаграммами. Кроме того, появились новые виды диаграмм: диаграммы обзора взаимодействия, временные диаграммы и диаграммы составных структур.

Значительное количество изменений не отражено в книге. Я не включил в обсуждение такие конструкции, как расширения конечных автоматов, шлюзы в диаграммах взаимодействия и типы мощности в диаграммах классов.

Поэтому в данном разделе рассказывается только об изменениях, вошедших в книгу «UML. Основы». Это либо изменения, о которых я рассказывал в предыдущих изданиях, либо новые, которые рассматриваются в этой книге. Поскольку изменения столь обширные, я расположил их в соответствии с организацией глав данного издания.

Диаграммы классов: основы (глава 3)

Атрибуты и ненаправленные ассоциации теперь представляют просто различные обозначения одного и того же базового понятия свойства. Дискретные кратности, такие как [2, 4], были исключены. Свойство `frozen` (замороженный) также было исключено. Я добавил перечень ключевых слов для обозначения общей зависимости, некоторые из которых появились только в UML 2. Ключевые слова «`parameter`» (параметр) и «`local`» (локальный) были выброшены.

Диаграммы последовательности (глава 4)

Значительно изменилась нотация фреймов взаимодействия, позволяя реализовывать различные сценарии управления поведением системы,

такие как итеративный, условный и другие. Теперь с помощью диаграммы последовательности можно довольно полно описать алгоритмы, хотя я не уверен, что программный код менее понятен. Применяемые ранее маркеры итерации и защиты в сообщениях были исключены из диаграммы последовательности. Заголовки линий жизни больше не представляют экземпляры классов; я называю их участниками (participants). Диаграммы, которые назывались в UML 1 диаграммами кооперации, в UML 2 называются коммуникационными диаграммами.

Диаграммы классов: дополнительные понятия (глава 5)

Определение стереотипов теперь стало более строгим. В результате теперь я рассматриваю слова в «кавычках» как ключевые слова, из которых лишь некоторые представляют собой стереотипы. Экземпляры на диаграммах объектов являются теперь спецификациями экземпляров. Классы теперь могут как требовать интерфейсы, так и предоставлять их. В случае множественной классификации обобщающие множества теперь служат для объединения обобщений в группы. Компоненты теперь не сопровождаются специальным символом. Активные объекты вместо жирной линии теперь обозначаются двойной вертикальной линией.

Диаграммы состояний (глава 10)

В UML 1 различали короткоживущие операции и долгоживущие деятельности. В UML 2 и то и другое называется деятельностями, но для долгоживущих деятельностей употребляется термин do-деятельность (do-activity).

Диаграммы деятельности (глава 11)

В UML 1 диаграммы деятельности рассматривались как особый случай диаграммы состояний. В UML 2 эта связь была разорвана, и в результате были исключены правила, согласно которым ветвления и объединения в диаграммах деятельности должны были находиться в соответствии. Поэтому они становятся более понятными при рассмотрении маркеров потоков, а не переходов состояний. Появилась целая группа новых нотаций, включая сигналы времени и приема, параметры, описания объединений, контакты, преобразования потоков, символы поддиаграмм, области расширения и окончания потоков.

Простое, но неудобное изменение заключается в том, что в UML 1 считалось, что несколько входящих в активность потоков неявно имеют слияние, в то время как в UML 2 предполагается, что они имеют объединение. По этой причине я рекомендую в диаграммах деятельности указывать слияния или объединения явным образом.

«Плавательные дорожки» теперь могут быть многомерными и в большинстве случаев называются разделами.

Библиография

1. Scott Ambler, *Agile Modeling*, Wiley, 2002.
2. Kent Beck, *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.
3. Kent Beck and Martin Fowler, *Planning Extreme Programming*, Addison-Wesley, 2000.
4. Kent Beck and Ward Cunningham, «A Laboratory for Teaching Object-Oriented Thinking», *Proceedings of OOPSLA 89*, 24 (10): 1–6. *http://c2.com/doc/oopsla89/paper.html*
5. Grady Booch, *Object-Oriented Analysis and Design with Applications, Second Edition*. Addison-Wesley, 1994.
6. Grady Booch, Jim Rumbaugh, and Ivar Jacobson, *UML User Guide*, Addison-Wesley, 1999.
7. Peter Coad and Edward Yourdon, *Object-Oriented Analysis*, Yourdon Press, 1991.
8. Peter Coad and Edward Yourdon, *Object-Oriented Design*, Yourdon Press, 1991.
9. Alistair Cockburn, *Agile Software Development*, Addison-Wesley, 2001.
10. Alistair Cockburn, *Writing Effective Use Cases*, Addison-Wesley, 2001.
11. Larry Constantine and Lucy Lockwood, *Software for Use,* Addison-Wesley, 2000.
12. Steve Cook and John Daniels, *Designing Object Systems: Object-Oriented Modeling with Syntropy*, Prentice-Hall, 1994.
13. Deepak Alur, John Crupi, and Dan Malks, *Core J2EE Patterns*, Prentice-Hall, 2001.
14. Ward Cunningham, «EPISODES: A Pattern Language of Competitive Development.» In *Pattern Languages of Program Design 2,* Vlissides, Coplien, and Kerth, Addison-Wesley, 1996, pp. 371–388.
15. Bruce Powel Douglass, *Real-Time UML*, Addison-Wesley, 1999.

16. Martin Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

17. Martin Fowler, «The New Methodology», *http://martinfowler.com/articles/newMethodology.html*

18. Martin Fowler and Matthew Foemmel, «Continuous Integration», *http://martinfowler.com/articles/continuousIntegration.html*

19. Martin Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

20. Martin Fowler, *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.[1]

21. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns:Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

22. Jim Highsmith, *Agile Software Development Ecosystems*, Addison-Wesley, 2002.

23. Luke Hohmann, *Beyond Software Architecture*, Addison-Wesley, 2003.

24. Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

25. Ivar Jacobson, Maria Ericsson, and Agneta Jacobson, *The Object Advantage: Business Process Reengineering with Object Technology*, Addison-Wesley, 1995.

26. Norm Kerth, *Project Retrospectives*, Dorset House, 2001

27. Anneke Kleppe, Jos Warmer, and Wim Bast, *MDA Explained,* Addison-Wesley, 2003.

28. Philippe Kruchten, *The Rational Unified Process: An Introduction*, Addison-Wesley, 1999.

29. Craig Larman, *Applying UML and Patterns*, 2d ed., Prentice-Hall, 2001.

30. Robert Cecil Martin, *The Principles, Patterns, and Practices of Agile Software Development,* Prentice-Hall, 2003.

31. Steve McConnell, *Rapid Development: Taming Wild Software Schedules,* Microsoft Press, 1996.

32. Steve Mellor and Marc Balcer, *Executable UML*, Addison-Wesley, 2002.

[1] Мартин Фаулер «Рефакторинг: улучшение существующего кода». – Пер. с англ. – СПб: Символ-Плюс, 2002.

33. Bertrand Meyer, *Object-Oriented Software Construction*. Prentice-Hall, 2000.

34. James Martin and James J. Odell, *Object-Oriented Methods: A Foundation (UML Edition)*, Prentice Hall, 1998.

35. Michael Pont, *Patterns for Time-Triggered Embedded Systems*, Addison-Wesley, 2001.

36. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

37. Douglas Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Archtecture Volume 2: Patterns for Concurrent and Networked Objects*, Wiley, 2000.

38. James Rumbaugh, *OMT Insights*, SIGS Books, 1996.

39. James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenzen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

40. James Rumbaugh, Ivar Jacobson, and Grady Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley, 1999.

41. Sally Shlaer and Stephen J. Mellor, *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, 1989.

42. Sally Shlaer and Stephen J. Mellor, *Object Lifecycles: Modeling the World in States*. Yourdon Press, 1991.

43. Jos Warmer and Anneke Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

44. Rebecca Wirfs-Brock and Alan McKean, *Object Design: Roles Responsibilities and Collaborations*. Prentice-Hall, 2003.

Алфавитный указатель

X

А

Б

В

Г

Д

З

И

К

М

Р

С

Т

У

Ф

Ш

Э

Я