UML 2
и Унифицированный процесс

ВТОРОЕ ИЗДАНИЕ

ПРАКТИЧЕСКИЙ ОБЪЕКТНО-ОРИЕНТИРОВАННЫЙ АНАЛИЗ И ПРОЕКТИРОВАНИЕ

ДЖИМ АРЛОУ
АЙЛА НЕЙШТАДТ

СИМВОЛ

UML 2
and the Unified Process

Practical Object-Oriented
Analysis and Design

Second Edition

Jim Arlow and Ila Neustadt

Addison-Wesley

UML 2
и Унифицированный процесс

Практический объектно-ориентированный анализ и проектирование

Второе издание

Джим Арлоу и Айла Нейштадт

Символ®

Санкт-Петербург — Москва
2008

Джим Арлоу и Айла Нейштадт

UML 2 и Унифицированный процесс, 2-е издание

Практический объектно-ориентированный анализ и проектирование

Перевод Н. Шатохиной

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*А. Пальваль*
Редактор	*А. Петухов*
Корректура	*О. Макарова*
Верстка	*Д. Орлова*

Арлоу Д., Нейштадт И.
UML 2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование, 2-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2007. – 624 с., ил.

ISBN-13: 978-5-93286-094-6
ISBN-10: 5-93286-094-4

Сегодня многие книги посвящены или UML, или Унифицированному процессу (Unified Process, UP), но не им обоим. Арлоу и Нейштадт заполнили этот пробел книгой, являющей собою замечательный синтез UML и UP. Здесь вы изучите методики объектно-ориентированного анализа и проектирования, синтаксис и семантику UML и соответствующие аспекты UP. Книга содержит точный и лаконичный обзор UML и UP с точки зрения ОО аналитика и проектировщика. В издании четко и понятно рассказано о практическом применении UML 2 на этапах анализа и проектирования Унифицированного процесса. Вы узнаете о роли моделирования в цикле разработки ПО, и эти знания помогут вам ответить на вопрос: как и когда использовать (или не использовать) UML, чтобы найти оптимальное решение для своего проекта. Авторы приводят множество примеров и дают рекомендации, бесценные для начинающих разработчиков моделей. Опытные ОО аналитики и проектировщики найдут в книге полезное руководство и справочник по UML 2.

ISBN-13: 978-5-93286-094-6
ISBN-10: 5-93286-094-4
ISBN 0-321-32127-8 (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 01.10.2007. Формат 70x100 1/16 . Печать офсетная.
Объем 39 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Отзывы о книге

«Стандарт UML 2 группы OMG очень систематично и основательно определяет UML, но в нем не хватает описания того, как применять UML 2 в реальном проекте. Вот где пригодится «UML 2 и Унифицированный процесс», 2-е издание. В книге ясно и доходчиво рассказывается о практическом применении UML 2. Изложение сопровождается множеством примеров и рекомендаций. Книга очень полезна даже тем, кто не работает с Унифицированным процессом. «UML 2 и Унифицированный процесс», 2-е издание – обязательная книга для новичков в UML 2 и полезное руководство и справочник для опытных профессионалов».

– Роланд Лейбандгут (Roland Leibundgut),
технический директор, Zühlke Engineering Ltd.

«Авторы очень подробно описывают конструктивные элементы UML и то, как они поддерживают Унифицированный процесс. Эта книга – хорошая отправная точка для организаций и специалистов, которые переходят к UP и нуждаются в понимании того, как обеспечить визуализацию различных аспектов в соответствии с UP».

– Эрик Найбург (Eric Naiburg)
менеджер по маркетингу, Desktop Products
IBM Rational Software

«Сегодня многие книги посвящены или UML, или Унифицированному процессу (Unified Process, UP), но не им обоим. Арлоу и Нейштадт заполнили этот пробел книгой, являющей собой замечательный синтез UML и UP. Авторы предлагают богатый опыт, бесценный для начинающих разработчиков моделей и опытных OO аналитиков и проектировщиков. Логическая структура, основанная на рабочих потоках UP, и особый стиль изложения с использованием диаграмм деятельностей в начале каждой главы существенно упрощают работу с книгой. Это издание должно быть всегда под рукой и у профессионалов, и у студентов».

– Исхан Де Силва (Ishan De Silva)
разработчик программного обеспечения
Millennium Information Technologies, Шри-Ланка

«Если вы ищете книгу с рецептами, почитайте что-нибудь другое. Эта книга заставит вас думать! В ней описываются все синтаксические элементы UML, но, что более важно, она дает практический совет, как и когда использовать (или не использовать) UML. Вы научитесь думать о роли моделирования в процессе разработки. Эти знания помогут вам ответить на вопрос: как и когда использовать UML, чтобы найти оптимальное решение для своего проекта. «UML 2 и Унифицированный процесс», 2-е издание подготовит вас к успешному применению UML».

– Джос Уормер (Jos Warmer)
Ordina System Integration & Development, Нидерланды

«Авторы создали книгу, объединяющую два важных предмета, UML и Унифицированный процесс. «UML 2 и Унифицированный процесс» – превосходный справочник по UML 2. Издание рассказывает о возможностях UML и о том, как применять его в дисциплинах анализа и проектирования Унифицированного процесса. Эта книга должна быть на столе у каждого профессионала».

– Гэри Поллис (Gary Pollice)
профессор, преподаватель вычислительной техники
Вустерский политехнический институт

Благодарности

Хотелось бы поблагодарить Фабрицио Феррандина (Fabrizio Ferrandina), Вольфганга Еммериха (Wolfgang Emmerich) и наших друзей из Zühlke Engineering за то, что они сподвигли нас на создание учебного курса, который стал основой этой книги. Особая благодарность Роланду Лейбандгуту (Roland Leibundgut) из Zühlke за его комментарии к главам, посвященным прецедентам, и Джосу Уормеру (Jos Warmer) и Тому Ван Корту (Tom VanCourt) за их глубокий анализ главы по OCL. Огромное спасибо и остальным техническим редакторам: Глен Форд (Glen Ford), Бергеру Меллер-Педерсену (Birger Møller-Pedersen), Робу Петтиту (Rob Pettit), Гэри Поллису (Gary Pollice), Исхану Де Силва (Ishan De Silva) и Фреду Васкиевичу (Fred Waskiewicz). Спасибо Сью и Девиду Эпштейнам (Sue, David Epstein) за жизненно важную нетехническую поддержку в течение всего проекта. Благодарим Энди Полса (Andy Pols), поделившегося с нами своими идеями по поводу прецедентов и процесса производства программного обеспечения. Наша благодарность сотрудникам издательства Addison-Wesley Ларе Вайсонг (Lara Wysong), Мэри Лу Нор (Mary Lou Nohr) и Ким Арни Малкахи (Kim Arney Mulcahy) за их замечательную работу над текстом и нашему редактору Мэри О'Брайан (Mary O'Brien). Спасибо семейству Нейштадт (Neustadt) за их терпение и Элу Томсу (Al Toms) за огромную поддержку. И конечно же, нашим котам, Гомеру, Падди и Мег, за многие часы сна на рукописях, которые наполнили их «неописуемым качеством».

И наконец, мы должны выразить признательность «Трем амигос» – Гради Бучу (Grady Booch), Джиму Рамбо (Jim Rumbaugh) и Айвару Джекобсону (Ivar Jacobson) – за их высококлассную работу над UML и UP, которым посвящена эта книга.

Предисловие

Об этой книге

Цель этой книги – показать процесс объектно-ориентированного (ОО) анализа и проектирования с помощью Унифицированного языка моделирования (Unified Modeling Language, UML) и Унифицированного процесса (Unified Process, UP).

UML представляет собой язык визуального моделирования для ОО моделирования. UP обеспечивает каркас процесса производства программного обеспечения, указывающий, как осуществлять ОО анализ и проектирование.

О UP можно говорить много. В книге представлены только аспекты, имеющие непосредственное отношение к работе ОО аналитика/проектировщика. За подробной информацией по другим деталям UP обращайтесь к [Rumbaugh 1] и другим указанным в библиографии книгам по UP.

Здесь приведено достаточное количество информации по UML и ассоциированным с ним методикам анализа и проектирования, что обеспечивает возможность эффективно применять моделирование в реальном проекте. Согласно Стивену Меллору (Stephen Mellor) [Mellor 1], существует три способа использования UML:

- UML как эскиз – это неформальный подход к UML, при котором используется схематическое изображение диаграмм, помогающее визуализировать программную систему. Это несколько схоже с наброском идеи на обратной стороне салфетки. Эскизы не представляют практически никакой ценности кроме их исходного применения, не сохраняются и в конце концов выбрасываются. Для создания неформальных эскизов обычно используют доску или инструментальные средства рисования, такие как Visio и PowerPoint (*www.microsoft.com*).
- UML как модель – это более формальный и точный подход, при котором UML используется для подробного описания программной системы. Это как набор архитекторских планов или чертеж машины. UML-модель активно поддерживается и становится важным поставляемым артефактом проекта. Этот подход требует использования настоящего инструментального средства моделирования, такого как Rational Rose (*www.rational.com*) или MagicDraw UML (*www.magicdraw.com*).
- UML как исполняемый проект – с помощью MDA (Model Driven Architecture – архитектура, управляемая моделью) UML-модели мо-

гут использоваться как язык программирования. Создается достаточно подробная UML-модель, и система может быть скомпилирована прямо из нее. Это самое формальное и точное применение UML и, по нашему мнению, это будущее разработки программного обеспечения. При таком подходе необходим UML-инструмент, поддерживающий MDA, такой как ArcStyler (*www.arcstyler.com*). Рассмотрение MDA выходит за рамки обсуждения этой книги, хотя мы касаемся его вкратце в разделе 1.4.

Основное внимание в книге сосредоточено на UML как модели. Представленные технические приемы также подойдут и для использования UML как исполняемого проекта. Изучив UML как модель, вы свободно сможете использовать UML как эскиз в случае необходимости.

Мы попытались сделать наше представление UML и UP максимально простым и доступным.

Условные обозначения

Чтобы упростить ориентирование по книге, каждая глава снабжена планом в форме диаграммы деятельностей UML. Эти диаграммы показывают деятельности чтения и порядок прочтения всех разделов. Диаграммы деятельности подробно рассматриваются в главе 14, а сейчас рис. 1 поможет разобраться с диаграммами планов глав.

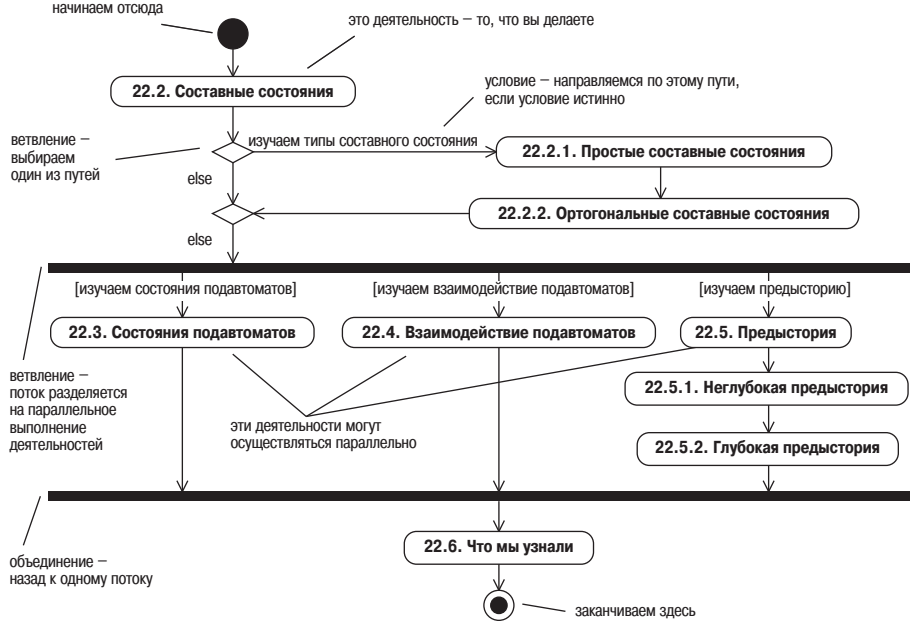


Рис. 1.

Большинство диаграмм в данной книге – это UML-диаграммы. Поясняющий текст на диаграммах не является частью синтаксиса UML.

> Важная информация оформлена в виде UML-пиктограммы примечания, т. е. заключена в прямоугольник с загнутым уголком.

В книге используются разные шрифты:

Этот шрифт применяется для элементов моделирования UML.

`Этот шрифт – для кода.`

Для кого эта книга

Мы видим следующих возможных читателей данной книги.

- Вы аналитик или проектировщик, которому необходимо научиться проводить ОО анализ и проектирование.
- Вы аналитик или проектировщик, которому необходимо научиться проводить ОО анализ и проектирование в рамках Унифицированного процесса.
- Вы студент, изучающий курс UML в университете.
- Вы разработчик программного обеспечения, которому необходима справочная информация по UML.
- Вы разработчик программного обеспечения, слушающий учебный курс по UML, и эта книга – ваш учебник.

Компания Clear View Training предлагает 4-дневный учебный курс по UML, основанный на данной книге. Этот курс читается по всей Европе нашим партнером, компанией Zuhlke Engineering (*www.zuhlke.com*), и доступен для лицензирования. Образовательные учреждения, использующие данную книгу как учебник, могут воспользоваться нашим учебным курсом бесплатно. Более подробно о коммерческом и учебном лицензировании см. по адресу *www.clearviewtraining.com*.

Как читать эту книгу

Так много книг и так мало времени, чтобы прочитать их все! Помня об этом, мы спланировали эту книгу так, что ее можно читать по-разному (в том числе и от корки до корки) соответственно вашим нуждам.

По ускоренной схеме

Выберите ускоренную схему, если хотите просто просмотреть всю книгу или отдельную главу. Кроме того, это способ получить сжатый смысл главы или книги.

- Выберите главу.
- Прочитайте план главы, чтобы знать, о чем пойдет речь.

- Просмотрите главу, останавливаясь на рисунках и примечаниях в рамках.
- Прочитайте раздел «Что мы узнали».
- Вернитесь и прочитайте любой заинтересовавший вас раздел.

Ускоренная схема – это быстрый и эффективный способ чтения этой книги. Возможно, вас приятно удивит, как много информации можно почерпнуть! Обратите внимание, что ускоренная схема эффективнее, если вы с самого начала можете четко сформулировать, какую информацию хотите получить. Например: «Я хочу понять, как осуществлять моделирование прецедентов».

Для справки

Если вам необходимо знать конкретную часть UML или изучить определенный технический прием, мы предоставили подробный индекс и оглавление, которые помогут найти необходимую информацию быстро и эффективно. Чтобы помочь в этом, в тексте используются точные перекрестные ссылки.

Просмотреть

Существует две стратегии просмотра данного текста.

- Если необходимо максимально эффективно и быстро освежить знания по UML, прочитайте краткие обзоры, приведенные в разделе «Что мы узнали» каждой главы. Если что-то непонятно, вернитесь и прочитайте соответствующий раздел.
- Если вы располагаете большим количеством времени, можно просмотреть каждую главу, изучая диаграммы и прочитывая примечания в рамках.

Пробежать глазами

Если у вас есть пара свободных минут, можно взять книгу и открыть ее на любой странице. Мы попытались сделать так, чтобы на каждой странице было что-то интересное. Если даже вы уже довольно хорошо знаете UML, все равно можно найти что-то новое.

План книги

На рис. 2 представлен план книги. Мы показали, какие главы можно читать в любом порядке, а какие можно пропустить при первом чтении, поскольку они обсуждают усовершенствованные технические приемы.

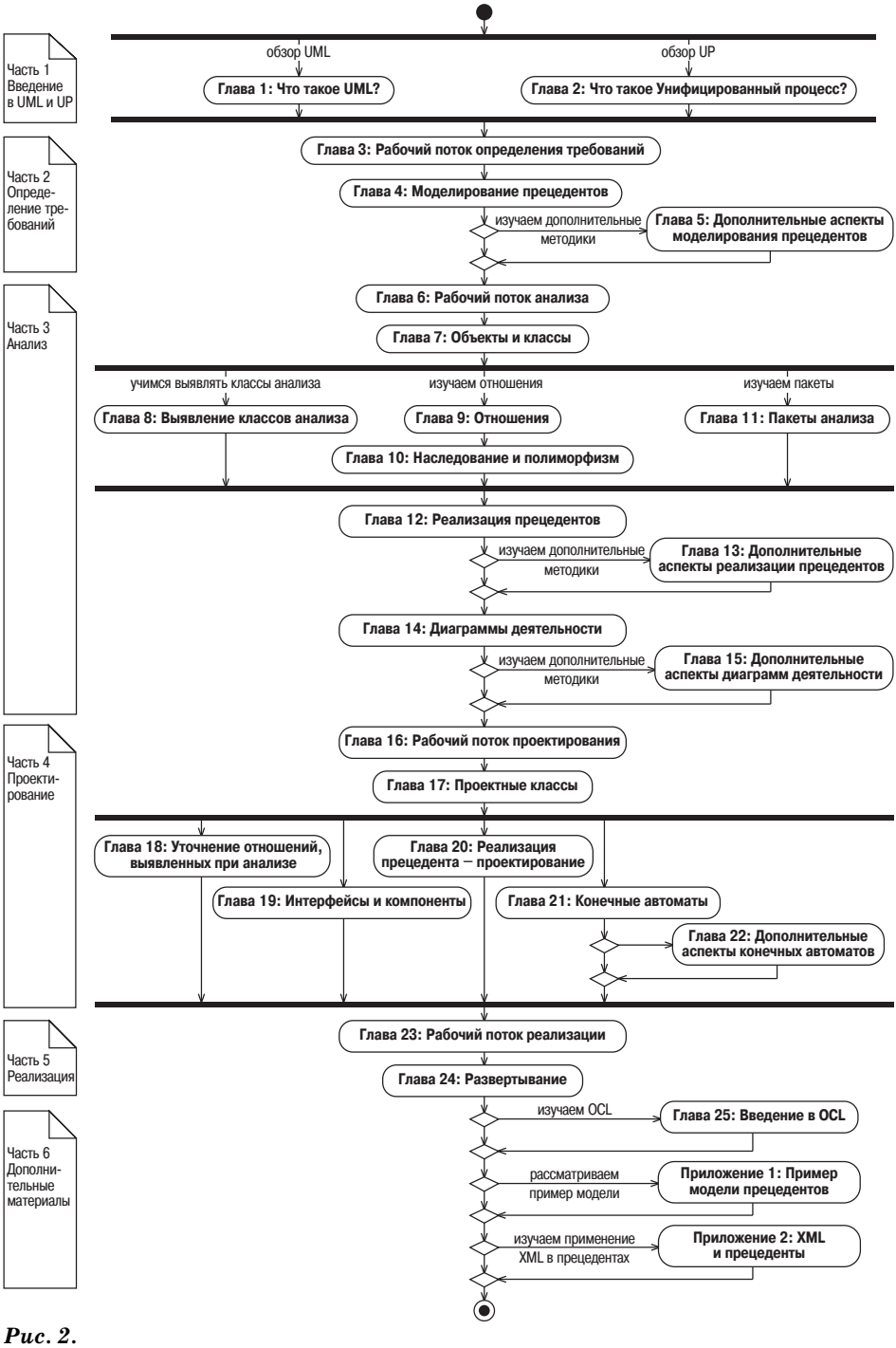

Часть 1 Введение в UML и UP
обзор UML
обзор UP
Глава 1: Что такое UML?
Глава 2: Что такое Унифицированный процесс?
Часть 2 Определение требований
Глава 3: Рабочий поток определения требований
Глава 4: Моделирование прецедентов
изучаем дополнительные методики
Глава 5: Дополнительные аспекты моделирования прецедентов
Часть 3 Анализ
Глава 6: Рабочий поток анализа
Глава 7: Объекты и классы
учимся выявлять классы анализа
изучаем отношения
изучаем пакеты
Глава 8: Выявление классов анализа
Глава 9: Отношения
Глава 11: Пакеты анализа
Глава 10: Наследование и полиморфизм
Глава 12: Реализация прецедентов
изучаем дополнительные методики
Глава 13: Дополнительные аспекты реализации прецедентов
Глава 14: Диаграммы деятельности
изучаем дополнительные методики
Глава 15: Дополнительные аспекты диаграмм деятельности
Часть 4 Проектирование
Глава 16: Рабочий поток проектирования
Глава 17: Проектные классы
Глава 18: Уточнение отношений, выявленных при анализе
Глава 20: Реализация прецедента – проектирование
Глава 19: Интерфейсы и компоненты
Глава 21: Конечные автоматы
Глава 22: Дополнительные аспекты конечных автоматов
Часть 5 Реализация
Глава 23: Рабочий поток реализации
Глава 24: Развертывание
Часть 6 Дополнительные материалы
изучаем OCL
Глава 25: Введение в OCL
рассматриваем пример модели
Приложение 1: Пример модели прецедентов
изучаем применение XML в прецедентах
Приложение 2: XML и прецеденты

Рис. 2.

I

Введение в UML и UP

1

Что такое UML?

1.1. План главы

В этой главе представлен краткий обзор истории развития UML и его структуры. Здесь перечислены темы, подробно рассматриваемые в последующих главах.

Те, кто не знаком с UML, должны начать с изучения его истории и основных принципов. Если вы уже имеете опыт работы с UML или убеждены, что достаточно знаете его историю, можете переходить прямо к разделу 1.7 и обсуждению структуры UML. Это обсуждение состоит из трех основных частей, которые можно читать в любом порядке. К ним относятся: строительные блоки UML (1.8), общие механизмы UML (1.9) и архитектура UML (1.10).

1.2. Что такое UML?

Унифицированный язык моделирования (Unified Modeling Language, UML) – это универсальный язык визуального моделирования систем. Хотя чаще всего UML ассоциируется с моделированием ОО программных систем, он имеет намного более широкое применение благодаря свойственной ему расширяемости.

UML объединил лучшие современные технические приемы моделирования и разработки программного обеспечения. По сути, язык UML был задуман так, чтобы его можно было реализовать посредством его же инструментальных средств. Фактически это признание того, что большие современные программные системы, как правило, нуждаются в инструментальной поддержке. UML-диаграммы легко воспринимаются и при этом без труда генерируются компьютерами.

Важно понимать, что UML *не* предлагает нам какой-либо методологии моделирования. Конечно, некоторые методические аспекты подразумеваются элементами, составляющими модель UML, но сам UML пре-

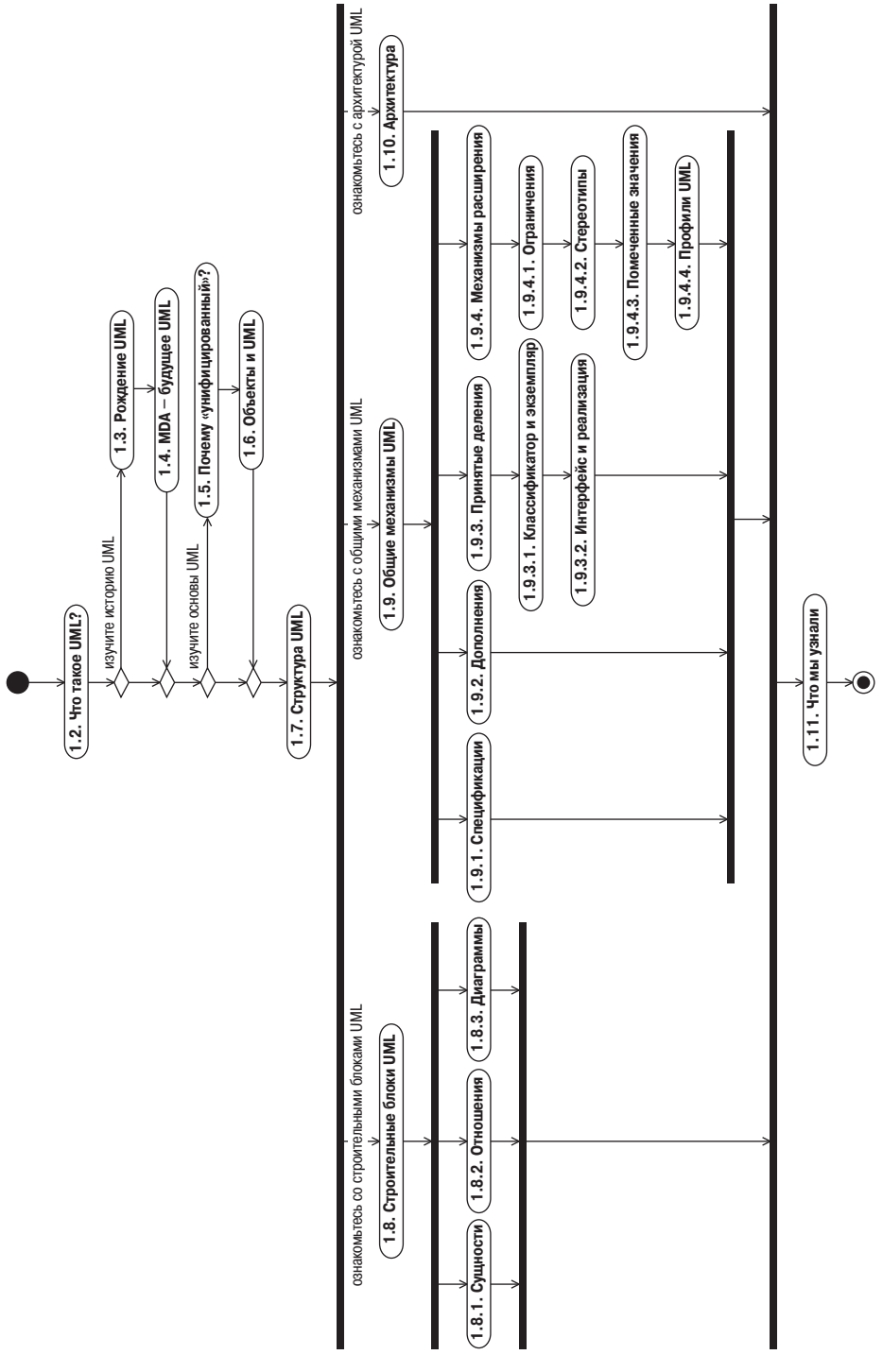


Рис. 1.1. План главы

доставляет собой лишь визуальный синтаксис, который можно использовать для создания моделей.

> UML это не методология, это унифицированный язык визуального моделирования. UP – это методология.

Унифицированный процесс (Unified Process, UP) – это методология. Она указывает на исполнителей, действия и артефакты, которые необходимо использовать, осуществить или создать для моделирования программной системы.

UML *не* привязан к какой-либо конкретной методологии или жизненному циклу. На самом деле он может использоваться со всеми существующими методологиями. UP использует UML в качестве базового синтаксиса визуального моделирования. Следовательно, UP можно рассматривать как *предпочтительный* метод для UML, поскольку он лучше всего адаптирован к нему. Однако сам UML способен предоставить (и предоставляет) поддержку визуального моделирования другим методам. Конкретным примером сложившегося метода, использующего UML в качестве визуального синтаксиса, является метод OPEN (Object-oriented Process, Environment, and Notation – объектно-ориентированный процесс, среда и нотация) (*www.open.org.au*).

Неизменная цель UML и UP – способствовать объединению всего лучшего в опыте разработки программного обеспечения последнего десятилетия. Для этого UML и UP *унифицируют* опыт предшествующих языков визуального моделирования и процессов разработки программного обеспечения наиболее оптимальным образом.

1.3. Рождение UML

До 1994 года в мире ОО методов царил хаос. Существовало несколько конкурирующих языков и методологий визуального моделирования, каждая с собственными преимуществами и недостатками, сторонниками и противниками. Среди языков визуального моделирования (рис. 1.2) очевидными лидерами были метод Буча (Booch Method), разработанный Гради Бучем (Grady Booch), и техника объектного моделирования (Object Modeling Technique, OMT) Джеймса Рамбо (James Rumbaugh), которые занимали более половины рынка. Что касается методологий, самую строгую систему создал Айвар Джекобсон (Ivar Jacobson). Несмотря на то, что многие авторы заявляли о создании «метода», фактически это были синтаксисы визуального моделирования и наборы более или менее полезных афоризмов и рекомендаций.

> UML – открытый, принятый в качестве промышленного стандарта язык визуального моделирования, одобренный OMG.

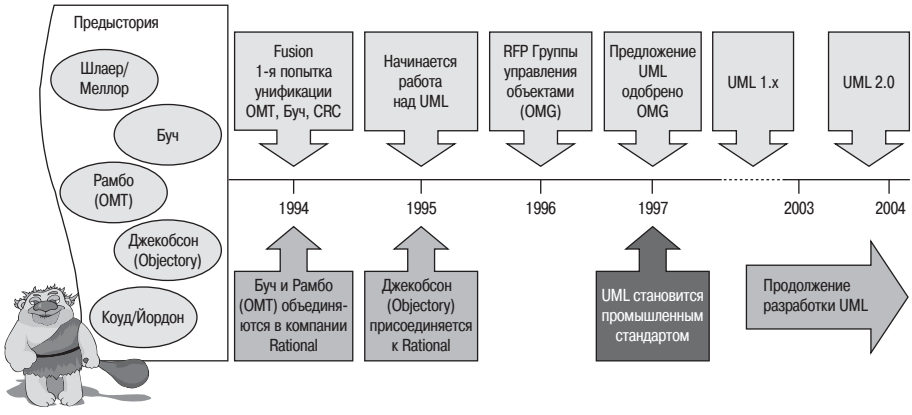


Рис. 1.2. *История UML*

Первую попытку унификации сделал в 1994 году Колеман (Coleman) в своем методе с использованием языка Fusion. Однако несмотря на все ее плюсы, к этой работе не были привлечены авторы-создатели составляющих методов (Буч, Джекобсон и Рамбо). К тому же книга с описанием этого подхода появилась на рынке слишком поздно. События развивались не в пользу метода с применением языка Fusion. В 1994 году Буч и Рамбо объединились в компании Rational Corporation для работы над UML. В то время это обеспокоило многих из нас, поскольку обеспечивало Rational более половины рынка по разработке подобного рода методов. Однако эти страхи оказались совершенно безосновательными, и UML стал открытым промышленным стандартом.

В 1996 г. Группа управления объектами (OMG, Object Management Group) выпускает запрос на предложение (request-for-proposal, RFP) для ОО языка визуального моделирования и предлагает UML. В 1997 г. OMG принимает UML и рождается первый открытый, удовлетворяющий промышленным стандартам ОО язык визуального моделирования. С этого момента исчезли все конкурирующие методы, и UML, бесспорно, стал стандартным ОО языком моделирования.

В 2000 году появилась версия UML 1.4 как существенное расширение UML, достигнутое добавлением семантики действий. Она описывает поведение набора элементарных действий, которые могут быть реализованы конкретными языками действий. Семантика действий плюс язык действий позволяют детально специфицировать поведенческие элементы модели UML (такие как операции классов) *непосредственно* в модели UML. Это было серьезным достижением, поскольку сделало спецификацию UML полной в вычислительном отношении, что обеспечило возможность делать UML-модели исполняемыми. Примером реализации UML, имеющей язык действий, совместимый с семантикой действий, является xUML, произведенный компанией Kennedy Carter (*www.kc.com*).

В 2005 году, когда еще шла работа над вторым изданием этой книги, была завершена спецификация UML 2.0. Теперь UML – вполне сформировавшийся язык моделирования. Прошло почти семь лет с момента выхода его первой версии, и он доказал свою ценность в сотнях проектах по разработке программного обеспечения по всему миру.

В UML 2 появилось много новых визуальных синтаксических структур. Некоторые из них замещают (и уточняют) существующий синтаксис версии 1.x, другие – абсолютно новые и представляют вновь введенную в язык семантику. UML как всегда предлагает множество вариантов представления конкретного элемента модели, но не все они будут поддерживаться каждым из инструментов моделирования. В этой книге мы пытаемся использовать лишь наиболее распространенные синтаксические варианты и обращаем внимание на те, которые полезны в обычных ситуациях моделирования. Некоторые синтаксические варианты слишком специализированны, поэтому они не обсуждаются или только упоминаются.

Хотя в UML 2 по сравнению с версией UML 1.x внесено множество синтаксических изменений, основополагающие принципы остались более или менее неизменными. Разработчики моделей, привыкшие к предыдущим версиям UML, легко перейдут на UML 2. Фактически самые глубокие изменения затронули метамодели UML, с которыми разработчики моделей непосредственно не будут иметь дело. Метамодель UML – это модель языка UML, выраженная в подмножестве UML. Она строго определяет синтаксис и семантику всех элементов моделирования UML, которые будут рассматриваться в книге. Изменения метамодели UML во многом касаются повышения точности и согласованности спецификации UML.

В одной из своих книг Гради Буч говорит: «Если у вас есть хорошая идея, она моя!» В этом заключена вся философия UML: он берет лучшее из того, что было до него, интегрирует и использует в качестве основы. Это можно понимать и в более широком смысле: UML объединяет лучшие идеи «доисторических» методов, отказываясь от их наиболее специфических деталей.

1.4. MDA – будущее UML

Будущее UML может быть определено, согласно недавнему предложению OMG, как архитектура, управляемая моделью (Model Driven Architecture, MDA). Хотя наша книга и не посвящена MDA, в этом разделе будет представлен ее краткий обзор. Более подробную информацию можно найти на веб-сайте OMG (*www.omg.org/mda*) и в книгах «MDA Explained» [Kleppe 1] и «Model Driven Architecture» [Frankel 1].

MDA дает видение того, как разрабатывать програмное обеспечение на основе моделей. Суть этого видения заключается в том, что модели управляют созданием исполняемой программной архитектуры. В настоящее время уже встречается подобный подход к разработке про-

граммного обеспечения, но MDA позволяет точно определить степень автоматизации данного процесса, чего до сих пор удавалось достичь довольно редко.

В MDA создание программного обеспечения происходит в результате ряда трансформаций модели при поддержке инструмента моделирования MDA. Абстрактная машинно-независимая модель (computer-independent model, CIM) используется как основа для платформонезависимой модели (platform-independent model, PIM). PIM трансформируется в платформозависимую модель (platform-specific model, PSM), которая преобразуется в код.

Понятие модели в MDA является довольно обобщенным, и код рассматривается как сильно конкретизированный вид модели. Рис. 1.3 иллюстрирует цепочку преобразований модели MDA.

CIM – это модель с очень высоким уровнем абстракции, фиксирующая ключевые требования системы и словарь предметной области *независимым* от компьютеров способом. Это действительно модель той части бизнес-процесса, которую предполагается автоматизировать. Такую модель создавать необязательно, и если она разрабатывается, то используется как основа для выработки PIM.

PIM – модель, выражающая семантику деятельности программной системы без ориентации на какую-либо базовую платформу (такую как EJB, .NET и т. д.). PIM обычно находится примерно на том же уровне абстракции, что и аналитическая модель, о которой пойдет речь несколько позже, но является более полной. Это является необходимым условием, поскольку данная модель должна обеспечивать достаточно полную базу для трансформации в PSM, из которой может быть сгенерирован код. В определении «платформонезависимый» немного смысла, пока точно не указаны платформы, от которых необходимо обеспечить независимость! Разные инструменты MDA поддерживают разные уровни независимости от платформы.

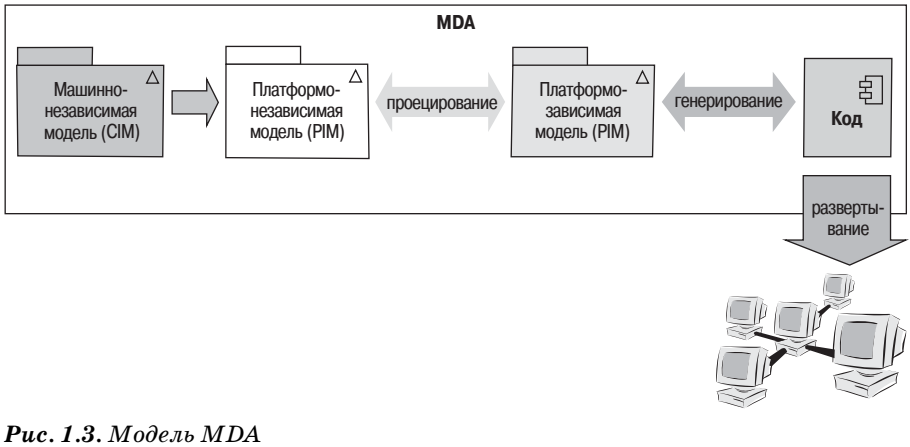


Рис. 1.3. *Модель MDA*

PIM дополняется характерной для конкретной платформы информацией для создания PSM. Из PSM генерируется исходный код под определенную платформу.

В принципе 100% исходного кода и вспомогательных артефактов, таких как документация, средства тестирования, файлы сборки и дескрипторы развертывания, могут генерироваться из достаточно полной PSM. Если это предполагается, модель UML должна быть *полной в вычислительном отношении*, иначе говоря, семантика всех операций должна быть определена на языке действий.

Как упоминалось ранее, некоторые инструментальные средства MDA уже предлагают язык действий. Например, инструмент iUML компании Kennedy Carter (*www.kc.com*) предоставляет язык спецификации действий (Action Specification Language, ASL), совместимый с семантикой действий UML 2. Этот язык действий более абстрактный, чем такие языки, как Java и C++, и может использоваться для создания полных в вычислительном отношении моделей UML.

Другие инструменты MDA, например ArcStyler (*www.io-software.com*), обеспечивают возможность генерировать от 70 до 90% кода и других артефактов, но тела операций должны быть дописаны на заданном языке программирования (например, Java).

В представлении MDA исходный код, такой как код на Java или C#, – это просто «машинный код», получающийся в результате компиляции моделей UML. Этот код генерируется в случае необходимости прямо из PSM. По существу, ценность кода при разработке с применением MDA значительно ниже, чем в UML-моделях. MDA превращает модели UML из прообраза создаваемого вручную исходного кода в основной механизм производства кода.

Во время подготовки книги к печати все больше и больше производителей инструментальных средств моделирования добавляли поддержку MDA в свои продукты. Самую свежую информацию можно найти на веб-сайте OMG, посвященном MDA. Существует также несколько многообещающих, использующих MDA инициатив с открытым исходным кодом, например Eclipse Modeling Framework (*www.eclipse.org/emf*) и AndroMDA (*www.andromda.org*).

В этом разделе мы ограничились «общей картиной» MDA. Спецификация MDA намного глубже, чем здесь было рассмотрено. Более подробную информацию можно найти по ссылкам, приведенным в начале этого раздела .

1.5. Почему «унифицированный»?

Унификация UML носит не только исторический характер. UML прилагает усилия (и в основном успешно) в унификации нескольких разных областей.

- Жизненный цикл разработки: UML предоставляет визуальный синтаксис для моделирования на протяжении всего жизненного цикла разработки программного обеспечения – от постановки требований до реализации.
- Области приложений: UML используется для моделирования всех аспектов – от аппаратных встроенных систем реального времени до систем поддержки принятия решений.
- Языки реализации и платформы: UML является независимым от языков и платформ. Естественно, он прекрасно поддерживает чистые ОО языки (Smalltalk, Java, C# и др.), но также эффективен и для гибридных ОО языков, таких как C++, и основанных на концепции объектов, таких как Visual Basic. UML также используется для создания моделей, реализуемых на неОО языках программирования, таких как C.
- Процессы разработки: хотя UP и его разновидности, вероятно, являются предпочтительными процессами разработки ОО систем, UML может поддерживать (и поддерживает) множество других процессов разработки ПО.
- Собственные внутренние концепции: UML поистине стойко стремится сохранить последовательность и постоянство применения небольшого набора своих внутренних концепций. До сих пор это не всегда удавалось, но в этом направлении наблюдается заметный прогресс по сравнению с предыдущими попытками.

1.6. Объекты и UML

Основная идея UML – возможность моделировать программное обеспечение и другие системы как *наборы взаимодействующих объектов*. Это, конечно же, замечательно подходит для ОО программных систем и языков программирования, но также очень хорошо работает и для бизнес-процессов и других прикладных задач.

В UML-модели есть два аспекта:

- Статическая структура – описывает, какие типы объектов важны для моделирования системы и как они взаимосвязаны.
- Динамическое поведение – описывает жизненные циклы этих объектов и то, как они взаимодействуют друг с другом для обеспечения требуемой функциональности системы.

Эти два аспекта модели UML идут рука об руку, и ни один из них не является по-настоящему полным без другого.

> UML моделирует мир как системы взаимодействующих объектов. Объект – это цельный блок, состоящий из данных и функциональности.

Объекты (и классы) будут подробно рассмотрены в главе 7. До тех пор будем считать, что объект единым целым блока данных и поведения. Иначе говоря, объекты содержат информацию и могут выполнять функции.

1.7. Структура UML

Понимание работы UML как визуального языка начинается с рассмотрения его структуры. Она показана на рис. 1.4 (как выяснится позже, это действительная UML-диаграмма). Эта структура включает:

- строительные блоки – основные элементы, отношения и диаграммы UML-модели;
- общие механизмы – общие UML-пути достижения определенных целей;
- архитектура – UML-представление архитектуры системы.

Понимание структуры UML дает нам представление о структуре всего изложенного в книге материала. Наличие структуры также указывает на то, что сам UML – это спроектированная система с собственной архитектурой. Кстати, UML был смоделирован и спроектирован с помощью UML! Этим проектом является метамодель UML.

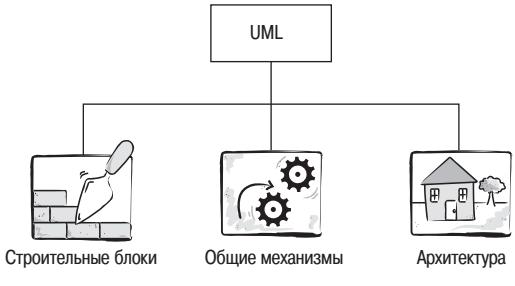


Рис. 1.4. *Структура UML*

1.8. Строительные блоки UML

Согласно «The Unified Modeling Language User Guide» [Booch 2], UML состоит всего из трех строительных блоков (рис. 1.5):

- Сущности – это сами элементы модели.
- Отношения связывают сущности. Отношения определяют, как семантически связаны две или более сущностей.
- Диаграммы – это *представления* моделей UML. Они показывают наборы сущностей, которые «рассказывают» о программной системе и являются нашим способом визуализации того, *что* будет делать система (аналитические диаграммы) или *как* она будет делать это (проектные диаграммы).

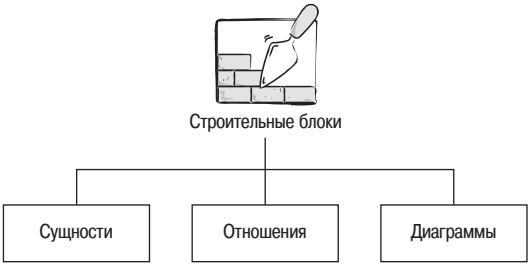


***Рис. 1.5.** Строительные блоки UML*

В следующих трех разделах типы строительных блоков рассматриваются немного подробнее.

1.8.1. Сущности

> «Сущности» – это существительные UML-модели.

Все UML-сущности можно разделить на:

- структурные сущности – существительные UML-модели, такие как класс, интерфейс, кооперация, прецедент, активный класс, компонент, узел;
- поведенческие сущности – глаголы UML-модели, такие как взаимодействия, деятельности, автоматы;
- группирующая сущность – пакет, используемый для группировки семантически связанных элементов модели в образующие единое целое модули;
- аннотационная сущность – примечание, которое может быть добавлено к модели для записи специальной информации, очень похожее на стикер.

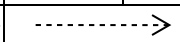
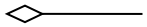
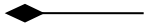
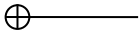
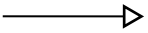
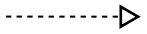
Все эти сущности и то, насколько успешно они используются в UML-моделировании, рассматривается в части 2.

1.8.2. Отношения

Отношения позволяют показать взаимодействие в пределах модели двух или более сущностей. Для понимания роли, которую отношения играют в моделях UML, достаточно представить отношения между членами отдельной семьи. Отношения в модели UML позволяют зафиксировать значимые (семантические) связи между сущностями. Например, в табл. 1.1 представлены UML-отношения, применяемые к структурным и группирующим сущностям модели.

Правильное понимание семантики различных типов отношений является очень важной частью моделирования UML. Однако мы отложим подробное описание этих семантик до следующих разделов книги.

Таблица 1.1

Тип отношения	UML-синтаксис		Краткая семантика	Раздел
	источник	цель		
Зависимость			Исходный элемент зависит от целевого элемента и изменение последнего может повлиять на первый.	9.5
Ассоциация			Описание набора связей между объектами.	9.4
Агрегация			Целевой элемент является частью исходного элемента.	18.4
Композиция			Строгая (более ограниченная) форма агрегирования.	18.5
Включение			Исходный элемент содержит целевой элемент.	11.4
Обобщение			Исходный элемент является специализацией более обобщенного целевого элемента и может замещать его.	10.2
Реализация			Исходный элемент гарантированно выполняет контракт, определенный целевым элементом.	12.3

1.8.3. Диаграммы

> Диаграммы – это только представления модели.

Во всех инструментальных средствах UML-моделирования новые сущности или новые отношения при создании добавляются в модель. Модель – это хранилище всех сущностей и отношений, созданных для описания требуемого поведения проектируемой программной системы.

Диаграммы – это своего рода *картины,* или *представления* модели. Диаграмма это *не* модель! На самом деле, различие между диаграммой и моделью является очень важным для понимания, поскольку сущность или отношение могут быть удалены с диаграммы, или даже со всех диаграмм, но по-прежнему они продолжают существовать в модели. Они будут оставаться в модели до тех пор, пока не будут явно удалены из нее. Общая ошибка новичков в UML-моделировании состоит в том, что они удаляют сущности с диаграмм, не удаляя их из модели.

Существует тринадцать различных типов UML-диаграмм, все они приведены на рис. 1.6. На рисунке каждый прямоугольник представляет определенный тип диаграммы, при этом курсивом выделяются абстрактные категории типов диаграмм. Например, существует шесть разных типов Диаграмм структуры. Обычный текст указывает на конкретную диаграмму, которую можно реально создать. Заштрихованные блоки обозначают типы диаграмм, впервые появившиеся в UML 2.

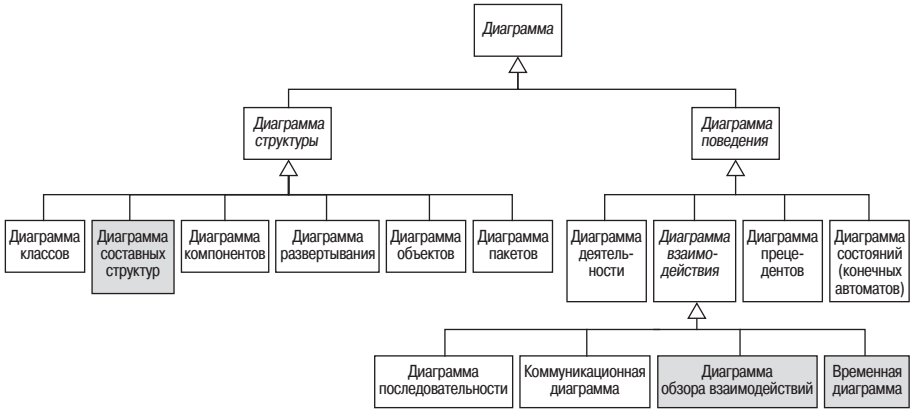


***Рис. 1.6.** Типы UML-диаграмм*

Эти диаграммы можно разделить на те, которые моделируют статическую структуру системы (статическую модель), и те, которые моделируют динамическую структуру системы (динамическую модель). Статическая модель фиксирует сущности и структурные отношения между ними; динамическая модель отображает, как сущности взаимодействуют для генерирования требуемого поведения программной системы. Статическая и динамическая модели рассматриваются начиная с части 2.

Определенного порядка создания UML-диаграмм не существует, хотя обычно начинают с диаграммы прецедентов для определения предметной области системы. Как правило, работа идет одновременно над несколькими диаграммами, каждая из которых уточняется по мере выявления более подробной информации о разрабатываемой программной системе. Таким образом, диаграммы являются как представлением модели, так и основным механизмом введения информации в модель.

В UML 2 представлен новый синтаксис диаграмм, изображенный на рис. 1.7. У каждой диаграммы может быть рамка, область заголовка и область содержимого. Область заголовка – это неправильный пятиугольник, содержащий тип (не обязательно), имя и параметры (не обязательно) диаграммы.

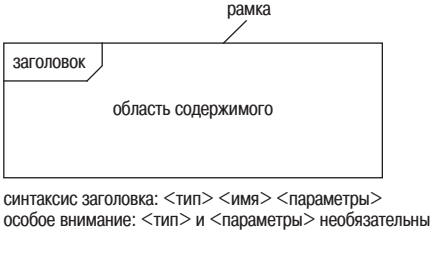


***Рис. 1.7.** Синтаксис UML-диаграмм*

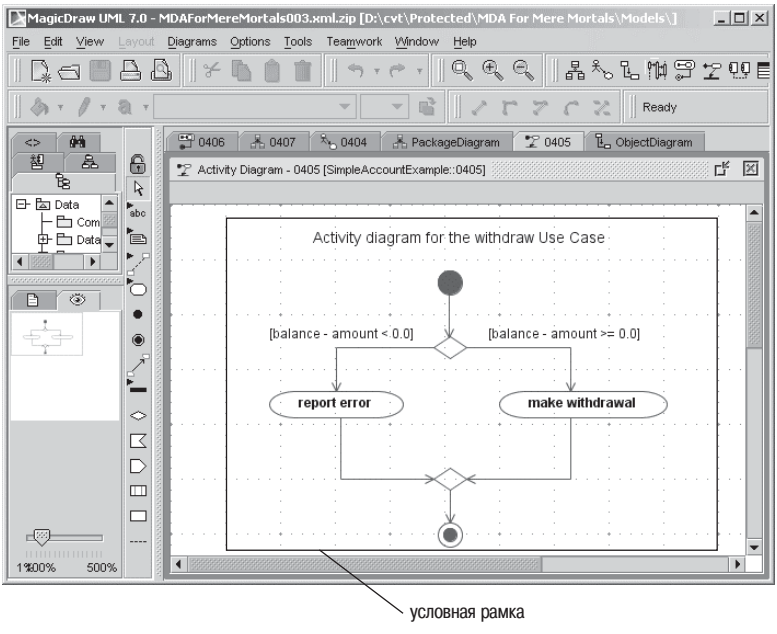


Рис. 1.8. *Пример диаграммы*

<Тип> указывает тип данной диаграммы и должен быть одним из типов, перечисленных на рис. 1.6. Спецификация UML определяет, что <тип> может быть сокращен, но не предоставляет списка стандартных сокращений. Явное задание <типа> требуется редко, потому что его обычно легко определить из визуального синтаксиса.

<Имя> должно описывать семантику диаграммы (например, CourseRegistration (регистрация курса)). <Параметры> предоставляют информацию, необходимую элементам модели, представленным на диаграмме. Примеры использования <параметров> будут приведены позже.

У диаграммы может быть (не обязательно) условная рамка, ограничивающая область в инструменте моделирования, внутри которой находится диаграмма. Пример условной рамки приведен на рис. 1.8.

1.9. Общие механизмы UML

В UML существует четыре общих механизма, последовательно применяемых ко всему языку моделирования. Они описывают четыре стратегии подхода к моделированию объектов, которые в разных контекстах многократно применяются в UML. Это еще раз убеждает нас в простоте и элегантности структуры UML (рис. 1.9).

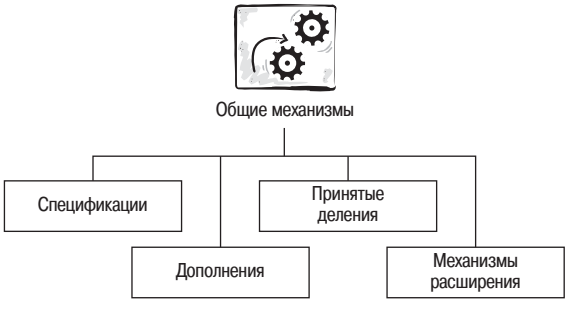


Рис. 1.9. *Общие механизмы UML*

1.9.1. Спецификации

> Спецификации – это суть UML-модели. Они обеспечивают семантический задний план модели.

Модели UML имеют, по крайней мере, два измерения: графическое, позволяющее визуализировать модель с помощью диаграмм и пиктограмм, и текстовое, состоящее из спецификаций различных элементов модели. Спецификации – это текстовые описания семантики элемента.

Класс, например BankAccount (банковский счет), можно изобразить в виде прямоугольника с ячейками (рис. 1.10), но это представление ничего не сообщает о бизнес-семантике этого класса. Семантика элементов модели фиксируется в спецификациях; без них можно только догадываться, что на самом деле представляет собой элемент.

Набор спецификаций – это суть модели. Спецификации формируют *семантический задний план* (*semantic backplane*), который объединяет модель и наполняет ее смыслом. Различные диаграммы – это просто представления или визуальные проекции этого плана.

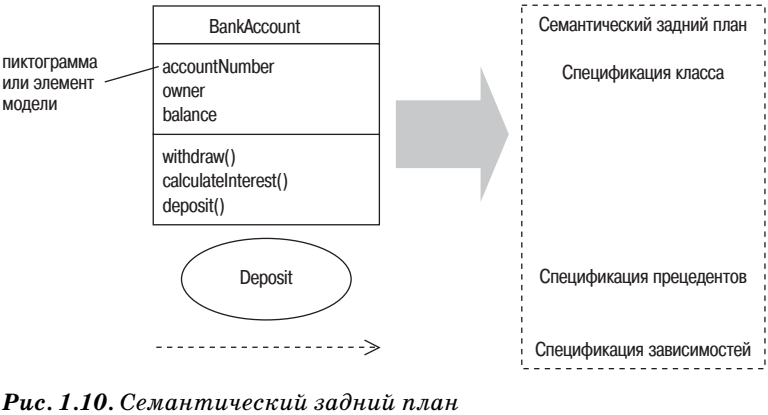


Рис. 1.10. *Семантический задний план*

Семантический задний план обычно сопровождается инструментом моделирования UML, предоставляющим доступ, просмотр и изменение спецификаций каждого элемента модели.

> Диаграммы обеспечивают представления семантического заднего плана.

UML обеспечивает большую гибкость при создании моделей. В частности, модели могут быть:

- сокращенными – некоторые элементы присутствуют в заднем плане, но скрыты в той или иной диаграмме для упрощения представления;
- неполными – некоторые элементы модели могут быть полностью пропущены;
- несогласованными – модель может содержать противоречия.

Здесь важен сам факт ослабления требований к полноте и согласованности, поскольку, как вы заметите, со временем модель эволюционирует и неоднократно подвергается изменениям. Однако развитие всегда происходит по направлению к *согласованным моделям*, *достаточно полным* для создания программной системы.

Разработку моделей с помощью UML, как правило, начинают с графической модели, которая позволяет визуализировать систему, а затем по мере ее развития добавляют в задний план все больше и больше семантики. Однако модель можно считать полезной или полной, только если семантика модели *присутствует* в семантическом заднем плане. В противном случае модели не существует, есть просто бессмысленный набор блоков и пятен, соединенных линиями! Кстати, общую ошибку, совершаемую новичками в разработке моделей, можно назвать «смерть от диаграмм»: модель переполнена диаграммами, но недоопределена.

1.9.2. Дополнения

В UML каждый элемент модели обозначается простым символом, к которому можно добавлять ряд дополнений, визуализирующих аспекты спецификации элемента. С помощью этого механизма видимая на диаграмме информация может быть представлена в соответствии с конкретными требованиями.

> Мы дополняем элементы модели на UML-диаграммах, чтобы подчеркнуть важные детали.

Начинать можно с создания высокоуровневой диаграммы, использующей только основные символы с одним или двумя дополнениями. Со временем диаграмма уточняется путем добавления все большего и большего количества дополнений до тех пор, пока не станет достаточно подробной.

Важно помнить, что любая UML-диаграмма – это только представление модели, и поэтому необходимо показывать лишь те дополнения, которые подчеркивают важные характеристики модели и делают диаграмму более понятной и удобочитаемой. Обычно нет необходимости показывать на диаграмме все до мельчайших подробностей. Важнее то, чтобы диаграмма была понятной, иллюстрировала именно те аспекты, которые требуется, и была легкой для восприятия.

На рис. 1.11 показано, что минимальным представлением класса является прямоугольник с именем класса. Однако различные детали базовой модели могут быть раскрыты в виде дополнений, расширяющих это минимальное представление. Возможные необязательные дополнения выделены на рисунке серым цветом.

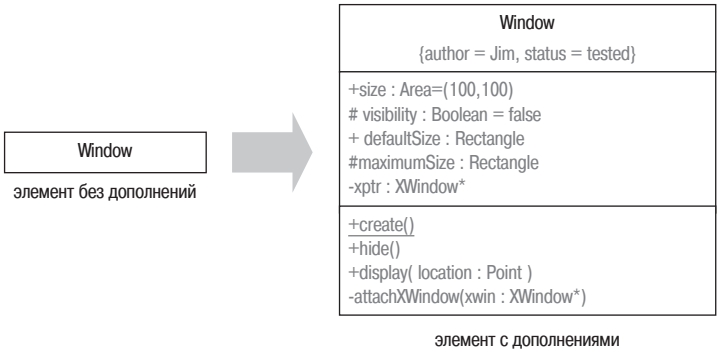


***Рис. 1.11.** Элемент с дополнениями*

1.9.3. Принятые деления

Принятые деления описывают конкретные способы представления мира. В UML существует два принятых деления: классификатор/экземпляр и интерфейс/реализация.

1.9.3.1. Классификатор и экземпляр

В UML предполагается, что может существовать абстрактное понятие типа сущности (например, банковский счет) и отдельные конкретные экземпляры этой абстракции (такие как «мой банковский счет» или «ваш банковский счет»). Абстрактное понятие типа сущности – это классификатор, а отдельные конкретные сущности – экземпляры. Это очень важная концепция, которая на самом деле крайне проста для понимания. Примеры классификаторов и экземпляров можно найти повсюду. Возьмем эту книгу по UML. Можно сказать, что «UML 2 и Унифицированный процесс» – это абстрактная идея этой книги, и множество ее экземпляров – это книги, одну из которых вы сейчас читаете. Вы увидите, что понятие «классификатор/экземпляр» является ключевой концепцией, пронизывающей UML.

В UML экземпляр обычно обозначается той же пиктограммой, что и соответствующий классификатор, но для экземпляров имя на пиктограмме подчеркнуто. Поначалу это различие может показаться несущественным.

> Классификатор – это абстрактное понятие, например тип банковского счета. Экземпляр – конкретная сущность, например ваш банковский счет или мой банковский счет.

UML 2 предоставляет богатую систематику из 33 классификаторов. Некоторые из наиболее распространенных классификаторов приведены в табл. 1.2.

Таблица 1.2

Классификатор	Семантика	Раздел
Актер	Роль, выполняемая внешним пользователем системы, которому система предоставляет некоторые услуги.	4.3.2
Класс	Описание набора объектов, обладающих одинаковыми свойствами.	7.4
Компонент	Модульная и замещаемая часть системы, инкапсулирующая свое содержимое.	19.8
Интерфейс	Набор операций, используемых для определения сервисов, предлагаемых классом или компонентом.	19.3
Узел	Физический элемент, существующий во время выполнения и представляющий собой вычислительный ресурс, например ПК.	24.4
Сигнал	Асинхронное сообщение, передаваемое между объектами.	15.6
Прецедент	Описание последовательности действий, осуществляемых системой для предоставления пользователю результата.	4.3.3

Все эти классификаторы (и другие) будут рассмотрены более подробно в следующих разделах.

1.9.3.2. Интерфейс и реализация

> Интерфейс – это, например, кнопки на панели видеомагнитофона. Реализация – устройство видеомагнитофона.

Основная идея этих понятий в том, чтобы отделить то, *что* выполняет действие (интерфейс), от того, *как* это делается (реализации). Например, при управлении машиной водитель взаимодействует с очень простым и четко определенным интерфейсом. На разных машинах этот интерфейс реализован по-разному.

Интерфейс определяет контракт (имеющий много общего с юридическим контрактом), придерживаться которого обязуются конкретные реализации. Это функциональное разделение между тем, что обещано выполнить, и реализацией этого обещания является важной концепцией UML. Подробно этот вопрос обсуждается в главе 17.

Конкретные примеры интерфейсов и реализаций можно найти повсюду. Например, кнопки на панели видеомагнитофона обеспечивают простой интерфейс его сложного механизма. Интерфейс избавляет нас от необходимости вникать в детали внутреннего устройства.

1.9.4. Механизмы расширения

> UML – расширяемый язык моделирования.

Разработчики UML понимали, что просто невозможно создать полностью универсальный язык моделирования, который удовлетворял бы всем современным требованиям и тем, что могут появиться в ближайшем будущем. Поэтому UML включает три простых механизма расширения, приведенные в табл. 1.3.

Таблица 1.3

	Механизмы расширения UML
Ограничения	Расширяют семантику элемента, обеспечивая возможность добавлять новые правила.
Стереотипы	Обеспечивают возможность определять новые элементы модели UML на основании существующих: мы определяем семантику стереотипа самостоятельно. Стереотипы добавляют новый элемент в метамодель UML.
Помеченные значения	Предоставляют способ расширения спецификации элемента, обеспечивая возможность добавлять в него новую специальную информацию.

Более подробно эти механизмы расширения рассматриваются в следующих трех разделах.

1.9.4.1. Ограничения

> Ограничения позволяют добавлять новые правила в элементы модели.

Ограничение – это строка текста, заключенная в фигурные скобки ({ }), определяющая некоторое условие или правило для элемента модели, которое *должно* оставаться истинным. Иначе говоря, оно некоторым образом ограничивает какие-либо свойства элемента. В книге приводятся примеры ограничений.

UML определяет Объектный язык ограничений (Object Constraint Language, OCL) как стандартное расширение. Введение в OCL изложено в главе 25.

1.9.4.2. Стереотипы

> Стереотипы позволяют определять новые элементы модели.

В книге «The UML Reference Manual» [Rumbaugh 1] утверждается: «Стереотип представляет разновидность существующего элемента модели, имеющего ту же форму (например, атрибуты и отношения), но другое назначение».

Стереотипы позволяют создавать новые элементы модели на основании *существующих*. Для этого к имени нового элемента добавляется имя стереотипа во французских кавычках («...»). Число стереотипов каждого элемента модели может изменяться от нуля до некоторого значения.

Каждый стереотип может определять ряд помеченных значений и ограничений, которые применяются к элементу, помеченному стереотипом. Также со стереотипом можно ассоциировать пиктограмму, цвет или текстуру. Обычно следует избегать применения цвета или текстуры в моделях UML, поскольку у некоторых читателей (например, дальтоников) могут возникнуть сложности с восприятием диаграмм. Кроме того, зачастую диаграммы распечатываются в черно-белом варианте. Обычно со стереотипом ассоциируют новую пиктограмму. Это позволяет контролировать расширение системы графических изображений UML.

Поскольку стереотипы вводят *новые* элементы модели с иным назначением, где-то должна быть определена семантика этих элементов. Как это сделать? Если инструмент моделирования не предоставляет встроенную поддержку документирования стереотипов, большинство разработчиков моделей просто помещают примечание в модель или вставляют ссылку на внешний документ, в котором описываются стереотипы. В настоящее время поддержка стереотипов инструментами моделирования не выполняется безоговорочно – большинство инструментальных средств поддерживают стереотипы в той или иной степени, но не все из них предоставляют возможность записи семантики.

С помощью элемента класс (глава 7) со специальным предопределенным UML-стереотипом «стереотип» можно самостоятельно моделировать стереотипы. При этом создается метамодель вашей системы стереотипов. Это метамодель, потому что она является моделью элементов модели и находится на совершенно другом уровне абстракции, чем обычная UML-система или бизнес-модели. Метамодель *ни в коем случае* нельзя объединять с обычными моделями. Она всегда должна находиться в отдельной модели. Создавать новую модель, предназначенную исключительно для стереотипов, имеет смысл, только если стерео-

типов много. Такая ситуация встречается довольно редко, поэтому большинство разработчиков моделей документируют стереотипы в примечаниях или внешних документах.

Стереотипы могут отображаться по-разному, но чаще всего разработчики моделей применяют просто имя стереотипа, заключенное в кавычки «», или пиктограмму. Другие варианты отображения стереотипов используются реже; кроме того, инструмент моделирования часто ограничивает возможности разработчика. Примеры приведены на рис. 1.12 (звездочки не являются частью UML-синтаксиса; они просто указывают на наиболее удачные варианты представления).

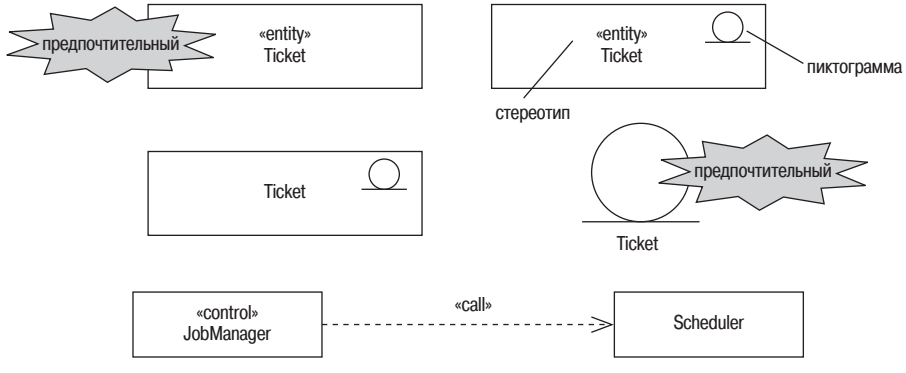


Рис. 1.12. *Варианты отображения стереотипов*

Обратите внимание, что помечать стереотипом можно и отношения. В книге вы найдете множество таких примеров.

1.9.4.3. Помеченные значения

> Помеченные значения позволяют добавлять собственные свойства к элементам модели.

В UML свойство – это любое значение, прикрепленное к элементу модели. Большинство элементов имеют большое число предопределенных свойств. Некоторые из них могут отображаться на диаграммах, другие являются частью семантического заднего плана модели.

С помощью помеченных значений UML позволяет добавлять в элементы модели собственные свойства. Идея помеченных значений предельно проста: помеченное значение – это ключевое слово, к которому может быть прикреплено значение. Помеченные значения имеют следующий синтаксис: {метка1 = значение1, метка2 = значение2, ..., меткаN = значениеN}. Разделенный запятыми список пар метка–значение, написанных через знак равенства, называется списком меток. Его заключают в фигурные скобки.

Некоторые метки являются просто дополнительной информацией об элементе модели, например {author = Jim Arlow} ({автор = Джим Арлоу}). Однако некоторые метки представляют свойства новых элементов модели, определенных стереотипом. Не следует применять такие метки непосредственно к элементам модели; лучше ассоциировать их со стереотипом. Тогда при применении стереотипа к элементу модели, элемент также получает метки, ассоциированные с этим стереотипом.

1.9.4.4. Профили UML

> Профиль UML определяет набор стереотипов, меток и ограничений, которые настраивают UML в соответствии с определенной целью.

Профиль UML – это набор стереотипов, помеченных значений и ограничений. Профиль UML используется, чтобы настроить UML для определенной цели.

Профили UML позволяют настраивать UML так, чтобы его можно было эффективно использовать в различных областях. Профили обеспечивают возможность согласованно и правильно применять стереотипы, метки и ограничения. Например, если UML используется для моделирования .NET-приложения, можно воспользоваться профилем UML для .NET, который приведен в табл. 1.4.

Таблица 1.4

Стереотип	Метки	Ограничения	Расширяет	Семантика
«NETComponent»	Нет	Нет	Компонент	Представляет компонент в .NET Framework.
«NETProperty»	Нет	Нет	Свойство	Представляет свойство компонента.
«NETAssembly»	Нет	Нет	Пакет	Упаковка компонентов времени выполнения .NET.
«MSI»	Нет	Нет	Артефакт	Файл автоматической установки компонента.
«DLL»	Нет	Нет	Артефакт	Переносимый исполняемый файл типа DLL.
«EXE»	Нет	Нет	Артефакт	Переносимый исполняемый файл типа EXE.

Этот профиль – один из примеров UML-профилей спецификации UML 2.0 [UML2S]. Он определяет новые элементы модели UML, адаптированные для моделирования .NET-приложений.

Каждый стереотип профиля расширяет один из элементов метамодели UML (например, Класс или Ассоциацию) для создания нового специального элемента. Стереотип может определять новые метки и ограничения, которые не являлись частью исходного элемента.

1.10. Архитектура

Книга «UML Reference Manual» [Rumbaugh 1] определяет архитектуру системы как «Организационную структуру системы, включая ее разбиение на части, их связность, взаимодействия, механизмы и направляющие принципы, передающие конструкцию системы». IEEE определяет архитектуру системы как «высокоуровневое представление системы в ее окружении».

> Стратегические аспекты системы можно описать «4+1 представлениями» архитектуры: логическое представление, представление процессов, представление реализации, представление развертывания и представление прецедентов.

Архитектура описывает фундаментальные аспекты высокоуровневой структуры системы. Существует множество способов описания архитектуры, но самым распространенным является «4+1 представление», данное Филиппом Крухтеном (Philippe Kruchten) [Kruchten 2]. Важнейшие аспекты архитектуры системы описываются четырьмя разными представлениями системы: логическим представлением, представлением процесса, представлением реализации и представлением развертывания. Все они интегрируются в пятое представление – представление прецедентов. Каждое из этих представлений раскрывает разные аспекты архитектуры программного обеспечения, как показано на рис. 1.13.

Давайте рассмотрим каждое из представлений по очереди.

- Логическое представление описывает словарь предметной области как набор классов и объектов. Основное внимание уделяется отображению того, как объекты и классы, образующие систему, реализуют требуемое поведение системы.
- Представление процессов моделирует исполняемые потоки и процессы системы как активные классы (классы, имеющие собственный поток управления). В действительности это процессориентированная версия логического представления, все их артефакты аналогичны.
- Представление реализации моделирует файлы и компоненты, образующие физическую базу из кода для системы. Это представление также иллюстрирует зависимости между компонентами и управляет конфигурированием наборов компонентов для определения версии системы.
- Представление развертывания моделирует физическое развертывание артефактов на физические вычислительные узлы, такие как компьютеры и периферийное оборудование. Оно обеспечивает возможность моделирования распределения артефактов между узлами распределенной системы.

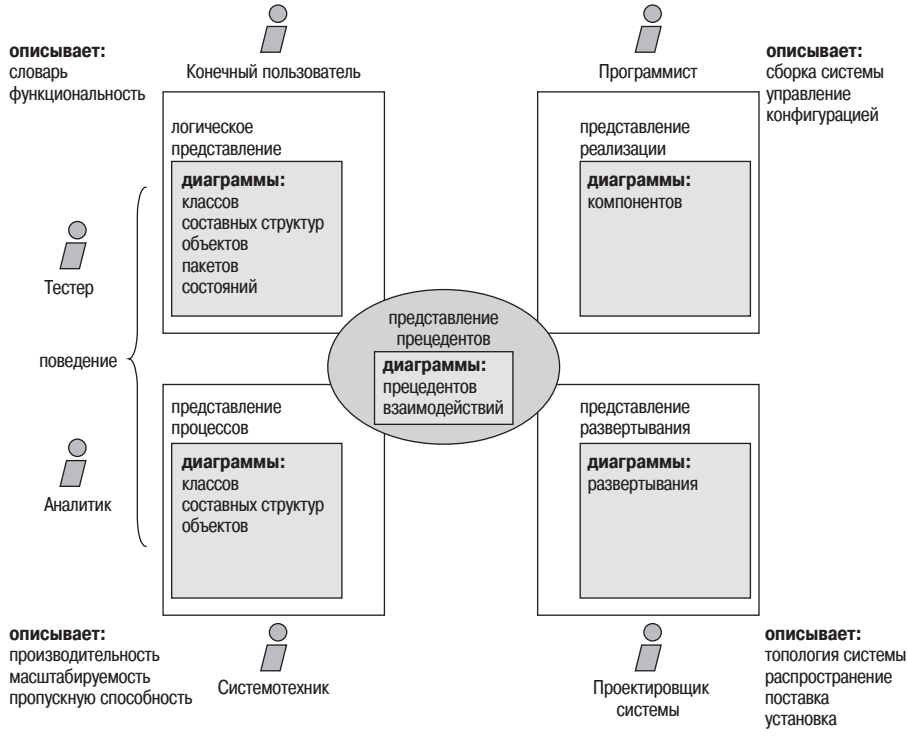


Рис. 1.13. *Архитектура системы. Адаптировано с рис. 5.1 [Kruchten 1] с разрешения издательства Addison-Wesley*

- Представление прецедентов описывает основные требования, предъявляемые к системе, как набор прецедентов (глава 4). Эти прецеденты обеспечивают базу для создания остальных представлений.

Как будет видно из дальнейшего изложения, UML предоставляет отличную поддержку каждого из 4+1 представлений, а UP является управляемым требованиями подходом, для которого замечательно подходит модель 4+1.

После создания 4+1 представлений необходимо с помощью UML-моделей проанализировать все ключевые аспекты архитектуры системы. Если используется итеративный жизненный цикл UP, архитектура 4+1 создается не за один проход, а развивается с течением времени. Процесс UML-моделирования с использованием UP – это процесс постепенного уточнения, приводящий к архитектуре 4+1, которая фиксирует как раз такой объем информации о системе, который обеспечивает возможность ее построения.

1.11. Что мы узнали

Эта глава содержит введение в историю, структуру, концепции и основные характеристики UML. Вы узнали следующее.

- Унифицированный язык моделирования (UML) – это открытый, расширяемый, принятый в качестве стандарта язык визуального моделирования, утвержденный консорциумом OMG.
- UML не методология.
- Унифицированный процесс (UP) или его разновидность – это тип методологии, наилучшим образом дополняющий UML.
- Объектное моделирование рассматривает мир как систему взаимодействующих объектов. Объекты содержат информацию и могут выполнять функции. UML-модели имеют:
  - статическую структуру – какие типы и объекты важны и как они взаимосвязаны;
  - динамическое поведение – как объекты взаимодействуют для осуществления функций системы.
- UML образован тремя строительными блоками:
  - сущности:
    - структурные сущности – существительные UML-модели;
    - поведенческие сущности – глаголы UML-модели;
    - существует только одна группирующая сущность – пакет, который используется для группировки семантически взаимосвязанных сущностей;
    - существует только одна аннотирующая сущность – примечание (аналог – стикер);
  - отношения объединяют сущности;
  - диаграммы показывают интересные представления модели.
- UML имеет четыре общих механизма:
  - спецификации – текстовые описания возможностей и семантики элементов модели, суть модели;
  - дополнения – элементы информации, обозначенные на элементе модели на диаграмме для пояснения какой-либо особенности;
  - принятые деления:
    - классификатор и экземпляр:
      - классификатор – абстрактное понятие типа сущности, например банковский счет;
      - экземпляр – конкретный экземпляр типа сущности, например мой банковский счет;
    - интерфейс и реализация:
      - интерфейс – контракт, определяющий поведение сущности;

    - реализация – конкретные детали того, как работает сущность;
- механизмы расширения:
  - ограничения позволяют добавлять новые правила для элементов модели;
  - стереотипы вводят новые элементы модели, базирующиеся на уже существующих;
  - помеченные значения позволяют добавлять новые свойства элементам модели; помеченное значение – это ключевое слово с ассоциированным значением;
  - профиль UML – набор ограничений, стереотипов и помеченных значений, позволяющий настроить UML для определенной цели.
- UML основывается на 4+1 представлениях архитектуры системы:
  - логическое представление – функциональность системы и словарь;
  - представление процессов – производительность, масштабируемость и пропускная способность системы;
  - представление реализации – сборка системы и управление конфигурацией;
  - представление развертывания – топология, распространение, поставка и установка системы;
  - все представления объединены представлением прецедентов, которое описывает требования заинтересованных сторон.

2

Что такое Унифицированный процесс?

2.1. План главы

В этой главе представлен краткий обзор Унифицированного процесса (Unified Process, UP). Новички должны начать с изучения истории UP. Если вам это уже известно, можно сразу перейти к разделу 2.4, где обсуждаются UP и RUP (Rational Unified Process, Унифицированный процесс компании Rational), или к разделу 2.5, в котором рассматривается возможность использования UP в вашем собственном проекте.

В этой книге нас интересует UP как каркас процесса, в рамках которого могут быть представлены технические приемы ОО анализа и разработки. Всестороннее обсуждение UP можно найти в книге [Jacobson 1], а замечательное описание RUP – в [Kroll 1], [Kruchten 2], а также в книгах [Ambler 1], [Ambler 2] и [Ambler 3].

2.2. Что такое UP?

Процесс производства программного обеспечения (Software Engineering Process, SEP), также известный как процесс разработки программного обеспечения (Software Development Process), определяет *кто*, *что*, *когда* и *как* в разработке ПО . Как показано на рис. 2.2, SEP – это процесс, в котором требования пользователя превращаются в ПО.

Унифицированный процесс разработки программного обеспечения (Unified Software Development Process, USDP) – это SEP от авторов UML. Обычно его называют Унифицированным процессом или UP [Jacobson 1]. В книге используется термин UP.

Проект UML должен был предоставить и визуальный язык, *и* процесс производства программного обеспечения. То, что сегодня называют UML, – это наглядная часть проекта, а UP – это процесс. Следует заметить, что UML был стандартизован OMG, UP – нет. Поэтому до сих пор не существует *стандартного* SEP для UML.

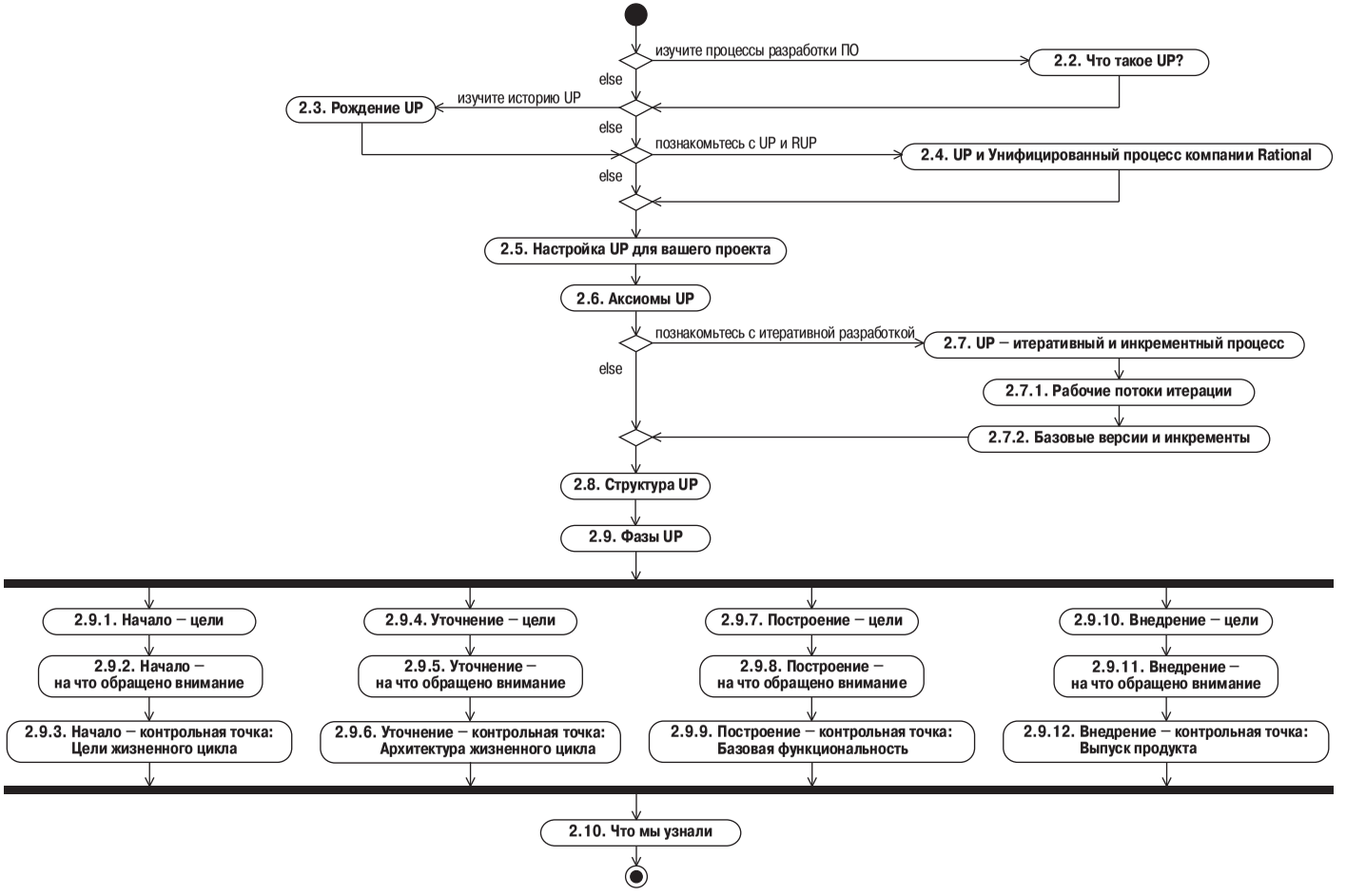


Рис. 2.1. *План главы*

Рис. 2.2. *Процесс производства ПО*

> Процесс производства программного обеспечения описывает то, как требования превращаются в программное обеспечение.

UP основывается на исследованиях процессов, проводимых в Ericsson (метод Ericsson, 1967), Rational (Rational Objectory Process, 1996–1997) и других ведущих компаниях. По сути, UP – внедренный в практику и проверенный метод разработки программного обеспечения, объединяющий в себе лучшие качества своих предшественников.

2.3. Рождение UP

Рассматривая историю UP, представленную на рис. 2.3, следует отметить, что его развитие тесно связано с карьерой одного человека, Айвара Джекобсона (Ivar Jacobson). Его часто называют отцом UP. Это никак не умоляет заслуги всех остальных специалистов (особенно Гради Буча), участвовавших в развитии UP, скорее, это подчеркивает значимость вклада Джекобсона.

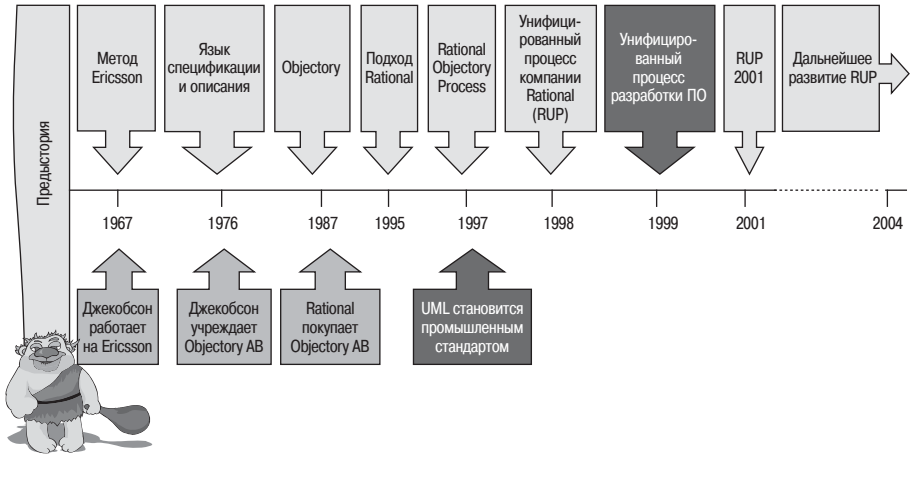


Рис. 2.3. *История UP*

> Работа над SEP, целью которой было развитие его до уровня UP, началась в 1967 г. в компании Ericsson.

Датой рождения UP можно считать 1967 год, а отправной точкой – подход Ericsson, который являлся радикальным шагом в моделировании сложных систем и рассматривал их как набор взаимосвязанных блоков. Меньшие блоки связывались между собой, образуя блоки большего размера, составляющие всю систему. Основополагающий принцип этого подхода – «разделяй и властвуй». Он является предтечей того, что сегодня известно как компонентно-ориентированная разработка.

Система, рассматриваемая как единое целое, может быть сложной для понимания. Для упрощения ее можно разбить на блоки меньшего размера и разобраться в предлагаемых каждым блоком сервисах (в современной терминологии – в интерфейсе компонента). Затем можно понять, как эти блоки совмещаются. В UML большие блоки называются подсистемами, и каждая подсистема реализуется меньшими блоками, называемыми компонентами.

Другим нововведением в подходе Ericsson был способ определения блоков через создание «вариантов трафика», описывающих, как предполагается использовать систему. Со временем варианты трафика эволюционировали, и в настоящее время в UML они называются прецедентами. В результате этого процесса появилось архитектурное представление, описывающее все блоки и их объединение. Это было предтечей статической модели UML.

Наряду с представлением требований (варианты трафика) и статическим представлением (описание архитектуры) у Ericsson было динамическое представление, описывающее, как все блоки взаимодействуют во времени. Это представление состояло из диаграмм последовательностей взаимодействия и состояний (конечных автоматов). Все они попрежнему есть в UML, хотя и в намного более совершенном виде.

Следующий крупный шаг в развитии производства ОО программного обеспечения был сделан в 1980 г. с выходом в свет Языка спецификации и описания (Specification and Description Language, SDL) от международной организации по стандартизации CCITT. SDL был одним из первых языков визуального моделирования, основанных на концепции объектов. В 1992 г. он был расширен и стал объектно-ориентированным языком с классами и наследованием. Этот язык был разработан для отображения поведения телекоммуникационных систем. Системы моделировались как набор блоков, общающихся друг с другом с помощью посылаемых сигналов. SDL-92 был первым широко принятым стандартом объектного моделирования, он используется и сегодня.

В 1987 г. Джекобсон основывает компанию Objectory AB в Стокгольме. Компанией был разработан и продан процесс производства ПО, основанный на методе Ericsson и названный Objectory (сокращение от «*Object* Fact*ory*» (фабрика объектов)). SEP компании Objectory включал

набор документации, уникальное инструментальное средство и необходимую консультацию, предоставляемую Objectory AB.

Вероятно, самым важным нововведением того времени было то, что сам по себе SEP компании Objectory рассматривался как система. Потоки работ (требования, анализ, разработка, реализация и тестирование) были представлены в виде набора диаграмм. Другими словами, процесс Objectory был смоделирован и разработан как программная система. Это проложило путь к будущей разработке процесса. Objectory, как и UP, был еще и каркасом (framework) процесса и требовал существенной доработки перед применением в конкретном проекте. Процесс Objectory поставлялся с несколькими шаблонами для различных типов проектов разработки программного обеспечения, однако ему неизменно требовалась существенная доработка. Джекобсон заметил, что все проекты по разработке программного обеспечения разные, и поэтому идея создания универсального SEP не была реальной и востребованной.

Когда в 1995 г. компания Rational приобрела Objectory AB, Джекобсон занялся объединением процесса Objectory с большим количеством наработок, выполненных в Rational. Было создано 4+1 представление архитектуры, базирующееся на четырех отдельных представлениях (логическом, процессов, физическом и разработки) плюс сводное представление прецедентов. Это до сих пор образует основу подхода UP к архитектуре системы. Кроме того, итеративная разработка была формализована в последовательность фаз (Начало, Уточнение, Построение, Внедрение), объединивших в себе упорядоченность водопадного жизненного цикла с динамизмом итеративной и инкрементной разработки. Основными создателями этой системы были Уолкер Ройс (Walker Royce), Рич Рейтман (Rich Reitmann), Гради Буч (Grady Booch) (создатель метода Буча) и Филипп Крухтен (Philippe Kruchten). В частности, опыт Буча и его идеи в отношении архитектуры были объединены в Rational Approach (подход компании Rational) (превосходное обсуждение его идей можно найти в [Booch 1]).

Rational Objectory Process (ROP) был результатом объединения подхода Objectory с исследованиями процессов компании Rational. В частности, ROP усовершенствовал области, в которых Objectory был слаб: требования, не входящие в прецеденты, реализация, тестирование, управление проектом, развертывание, управление конфигурацией и среда разработки. Было введено понятие риска (risk) как управляющего механизма ROP, а «архитектура» получила точное определение как поставляемое «архитектурное представление». В это время в компании Rational Буч, Джекобсон и Рамбо разрабатывали UML. Он стал языком, в котором были представлены модели ROP и сам ROP.

Начиная с 1997 г. Rational присоединила множество компаний, объединив опыт в определении требований, управлении конфигурацией, тестировании и т. д. Это привело к выходу в 1998 г. Унифицированного процесса компании Rational (Rational Unified Process, RUP). С тех

пор свет увидели множество версий RUP, каждая из которых неизменно лучше предыдущих. Более подробную информацию можно найти по адресу *www.rational.com* и в книге [Kruchten 1].

> UP – это сложившийся открытый SEP от авторов UML.

В 1999 г. была опубликована важная книга «Unified Software Development Process» [Jacobson 1], описывающая Унифицированный процесс. Если RUP – это процесс, являющийся продуктом компании Rational, то UP – открытый SEP от авторов UML. Не удивительно, что UP и RUP тесно взаимосвязаны. В нашем изложении мы решили использовать UP, а не RUP, поскольку он является открытым SEP, доступным для всех и не привязанным к конкретному продукту или производителю.

2.4. UP и Унифицированный процесс компании Rational

> RUP – это коммерческий продукт, расширяющий UP.

Унифицированный процесс компании Rational (Rational Unified Process, RUP) – это коммерческая версия UP от IBM, которая поглотила Rational Corporation в 2003 году. Он предоставляет стандарты, инструментальные средства и остальные необходимые элементы, которые не включены в UP и которые пользователям в противном случае пришлось бы искать самостоятельно. Он также поставляется с богатым веб-окружением, включающим всю документацию процесса и полные руководства для каждого инструментального средства.

> У процессов UP и RUP больше общего, чем отличий.

В 1999 г. RUP был практически прямой реализацией UP. С того времени RUP активно развивался. В настоящее время он расширяет UP во многих важных направлениях. Сегодня UP необходимо рассматривать как открытый обобщенный продукт, а RUP – как специальный коммерческий подкласс, который одновременно и расширяет, и переопределяет возможности UP. Но в RUP и UP по-прежнему остается больше общего, чем различного. Главное их отличие не в семантике или идеологии, а в полноте и детализации. Основные рабочие потоки ОО анализа и разработки совершенно аналогичны, поэтому описание с точки зрения UP будет также полезно и для пользователей RUP. Приняв решения использовать в нашей книге UP, мы сделали материал применимым для большинства ОО аналитиков и разработчиков, которые не работают с RUP, а также для небольшой, но существенной и постоянно увеличивающейся группы его пользователей.

И UP, и RUP моделируют *кто*, *когда* и *что* в процессе разработки программного обеспечения, но делают это немного по-разному. Самая последняя версия RUP имеет некоторые отличия от UP в терминологии и синтаксисе, хотя семантика элементов процесса остается, по сути, прежней.

На рис. 2.4 показано, каким пиктограммам RUP соответствуют пиктограммы UP, используемые в книге. Обратите внимание, что между пиктограммами RUP и оригинальными пиктограммами UP установлено отношение «trace» (след). В UML отношение «trace» является особым типом зависимости между элементами модели, указывающим на то, что элемент-источник отношения «trace» является результатом исторического развития элемента, на который указывает стрелка. Это идеальное описывает отношения между элементами моделей UP и RUP.

Для моделирования SEP-понятия «кто» UP вводит концепцию исполнителя (worker). Она описывает роль, исполняемую в проекте отдельным индивидуумом или командой. Каждый исполнитель может быть реализован множеством индивидуумов или команд, а каждый индивидуум или команда могут действовать как множество разных исполнителей. В RUP исполнители называются «ролями», но семантика остается той же.

UP моделирует понятие «что» в виде деятельностей и артефактов. Деятельности – это задачи, которые будут выполняться в проекте отдель-

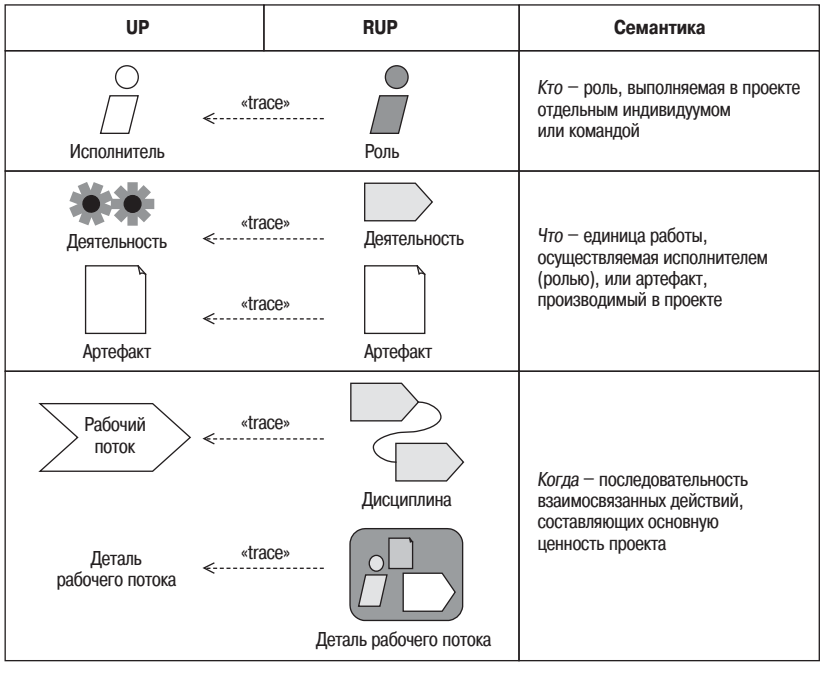


Рис. 2.4. *Соответствие пиктограмм UP и RUP*

ными индивидуумами или командами. При осуществлении определенной деятельности эти индивидуумы или команды всегда будут *принимать конкретные роли*. Поэтому для любой деятельности UP (и RUP) может указать исполнителей (роли), участвующих в ней. В случае необходимости деятельности могут быть разбиты на более мелкие уровни детализации. Артефакты – это сущности, являющиеся входными данными и результатами проекта. Это может быть исходный код, исполняемые программы, стандарты, документация и т. д. Для обозначения артефактов используются различные пиктограммы в зависимости от типа артефакта. На рис. 2.4 приведена универсальная пиктограмма документа.

UP моделирует понятие «когда» в виде рабочих потоков, которые представляют собой последовательности взаимосвязанных деятельностей, осуществляемых исполнителями. В RUP рабочим потокам высокого уровня, таким как Требования или Тестирование, присвоено специальное имя – *дисциплины* (*disciplines*). Рабочие потоки могут быть разбиты на одну или более составляющих, которые описывают деятельности, роли и артефакты, участвующие в рабочем потоке. Эти составляющие рабочего потока в UP обозначаются только по имени, а в RUP они имеют собственные пиктограммы.

2.5. Настройка UP для вашего проекта

> UP и RUP должны настраиваться под каждый конкретный проект.

UP – универсальный процесс разработки ПО, который должен настраиваться для определенной организации и для каждого конкретного проекта. Тем самым признается, что все проекты по разработке ПО разные и что подход «один размер для всех» в случае с SEP не работает. Процесс настройки включает определение и объединение:

- внутренних стандартов;
- шаблонов документов;
- инструментальных средств – компиляторов, инструментов управления конфигурацией и т. д.;
- баз данных – отслеживание ошибок, слежение за проектом и т. д.;
- изменений жизненного цикла – например более сложные меры контроля качества для систем с особыми требованиями к обеспечению безопасности.

Детали процесса настройки выходят за рамки рассмотрения книги, но их можно найти в [Rumbaugh 1].

Даже несмотря на то, что RUP намного более полный, чем UP, он все же должен подвергаться такой же подгонке и настройке. Однако необходимый для этого объем работы оказывается намного меньшим, чем если начинать с исходного UP. Кстати, для *любого* процесса производства

программного обеспечения нужно ожидать определенных затрат времени и средств на его настройку. Возможно придется обратиться за помощью к поставщику SEP и оплатить услуги консультанта.

2.6. Аксиомы UP

UP базируется на трех аксиомах. Он является:

- управляемым требованиями и риском;
- архитектуро-центричным;
- итеративным и инкрементным.

Прецеденты будут подробно рассмотрены в главе 4. Здесь стоит заметить, что прецеденты – это способ записи требований. Таким образом, можно с уверенностью утверждать, что UP является управляемым требованиями.

> UP – это современный SEP, который управляется требованиями пользователя и риском.

Риск – это еще один управляющий механизм UP, поскольку если не атаковать риски, они будут атаковать вас! Все, кто участвовал в проектах по разработке ПО, без сомнения, согласятся с этим утверждением. UP решает эту проблему, заложив анализ рисков в основу создания ПО. Вообще говоря, решение этой проблемы – дело руководителя проекта и архитектора, поэтому не будем останавливаться на этом подробно.

Подход UP к разработке программных систем заключается в создании и развитии надежной архитектуры системы. Архитектура описывает стратегические аспекты разбиения системы на компоненты, а также взаимодействия и развертывания этих компонентов на аппаратных средствах. Очевидно, что качественно разработанная архитектура обеспечит создание работоспособной системы, а не просто наспех скомпонованного набора кустарного исходного кода.

И наконец, UP является итеративным и инкрементным. Итеративный аспект UP означает, что проект разбивается на небольшие подпроекты (итерации), которые обеспечивают функциональность системы по частям, или инкрементам, приводя к созданию полнофункциональной системы. Другими словами, ПО создается путем пошаговой детализации. Такой подход сильно отличается от старого водопадного жизненного цикла. Последний включал в себя анализ, разработку и построение, которые следовали в более или менее прямой последовательности друг за другом. Фактически к ключевым рабочим потокам UP, таким как анализ, мы возвращаемся по нескольку раз в течение проекта.

2.7. UP – итеративный и инкрементный процесс

Чтобы понять UP, необходимо понять итерации. Идея, по существу, крайне проста: история показывает, что людям, вообще говоря, решать маленькие проблемы легче, чем большие. Поэтому мы разбиваем большой проект по разработке ПО на несколько меньших «мини-проектов», которыми легче управлять и успешно выполнить. Каждый из этих «мини-проектов» и есть итерация. Основной момент: каждая итерация включает *все* элементы обычного проекта по разработке ПО:

- Планирование
- Анализ и проектирование
- Построение
- Интеграция и тестирование
- Версия для внутреннего или внешнего использования

> Цель UP – пошагово построить надежную архитектуру системы.

Каждая итерация создает базовую версию, включающую в себя *частично завершенную* версию целевой системы и документацию проекта. В ходе последовательных итераций базовые версии наращиваются до тех пор, пока не будет создан окончательный полный вариант системы. Разница между двумя последующими базовыми версиями и есть инкремент. Вот почему UP называют итеративным и инкрементным жизненным циклом.

Как мы увидим в разделе 2.8, итерации группируются в фазы. Фазы образуют макроструктуру UP.

2.7.1. Рабочие потоки итерации

> В UP пять основных рабочих потоков.

В каждой итерации пять основных рабочих потоков определяют, что должно быть сделано и какие навыки для этого необходимы. Наряду с пятью основными рабочими потоками будут присутствовать и другие, такие как планирование, оценка и все, что характерно для этой конкретной итерации. Однако эти этапы не выделены в UP.

К пяти основным рабочим потокам относятся:

- определение требований – сбор данных о том, что должна делать система;
- анализ – уточнение и структурирование требований;
- проектирование – реализация требований в архитектуре системы;
- реализация – построение программного обеспечения;
- тестирование – проверяется, отвечает ли реализация предъявляемым требованиям.

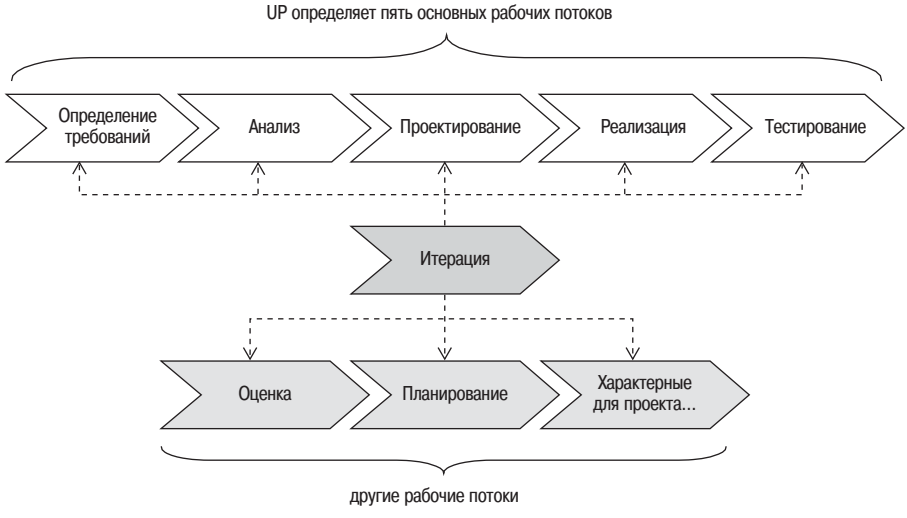


Рис. 2.5. *Возможные рабочие потоки итерации*

Некоторые возможные рабочие потоки итерации изображены на рис. 2.5. Более подробно потоки определения требований, анализа, проектирования и реализации (тестирование выходит за рамки обсуждения) будут рассмотрены несколько позже.

Хотя в каждой итерации могут присутствовать все пять рабочих потоков, в зависимости от местоположения итерации в жизненном цикле проекта внимание может быть акцентировано на каком-либо одном рабочем потоке.

Разбиение проекта на серию итераций обеспечивает возможность гибкого подхода к его планированию. Самый простой подход – упорядоченная во времени последовательность итераций, в которой каждая последующая итерация является результатом предыдущей. Однако часто итерации можно расположить параллельно. Это предполагает понимание зависимостей между артефактами каждой итерации и требует основанного на архитектуре и моделировании подхода к разработке ПО. Преимущество параллельных итераций – меньшее время вывода нового изделия на рынок и, возможно, более рациональное использование команды, но при этом первостепенным является тщательное планирование.

2.7.2. Базовые версии и инкременты

Каждая итерация UP формирует базовую версию. Это версия для внутреннего (или внешнего) использования набора рассмотренных и утвержденных артефактов, сгенерированных в данной итерации. Каждая базовая версия:

- предоставляет базу для дальнейшего рассмотрения и разработки;

- может изменяться *только* через формальные процедуры управления конфигурацией и изменениями.

Инкременты – это просто *разница* между двумя последовательными базовыми версиями. Это шаги по направлению к окончательной выпускаемой системе.

2.8. Структура UP

> В UP четыре фазы, каждая из которых имеет свои контрольные точки.

На рис. 2.6 показана структура UP. Жизненный цикл проекта разделен на четыре фазы: Начало, Уточнение, Построение и Внедрение, каждая из которых имеет свои контрольные точки. В каждой фазе может быть одна или более итераций, в каждой итерации выполняются пять основных и любое количество дополнительных рабочих потоков. Точное число итераций в фазе зависит от размера проекта, но каждая итерация должна длиться не более двух-трех месяцев. Приведенный пример является типовым для среднего проекта продолжительностью около 18 месяцев.

Как видно из рис. 2.6, UP состоит из последовательности четырех фаз, каждая из которых завершается важной контрольной точкой:

- Начало (Inception) – цели жизненного цикла;
- Уточнение (Elaboration) – архитектура жизненного цикла;
- Построение (Construction) – базовая функциональность;
- Внедрение (Transition) – выпуск продукта.

По мере прохождения этих фаз UP объем работы, выполняемый в каждом из пяти рабочих потоков, меняется.

Рисунок 2.7 – ключ к пониманию принципа работы UP. Вверху показаны фазы. В крайнем левом столбце – пять основных рабочих потоков. Внизу изображены итерации. Кривые показывают относитель-

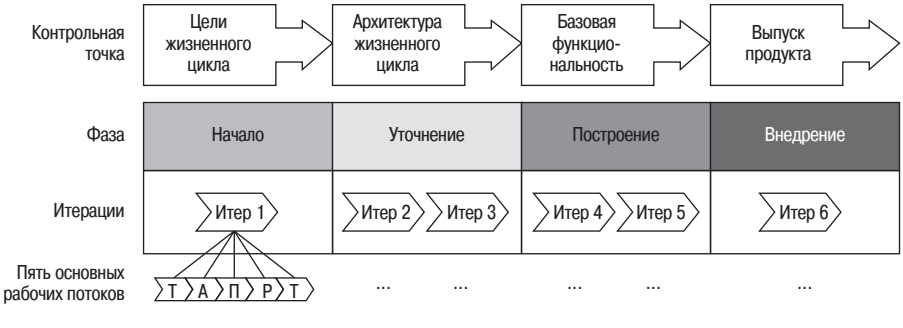


Рис. 2.6. *Структура UP*

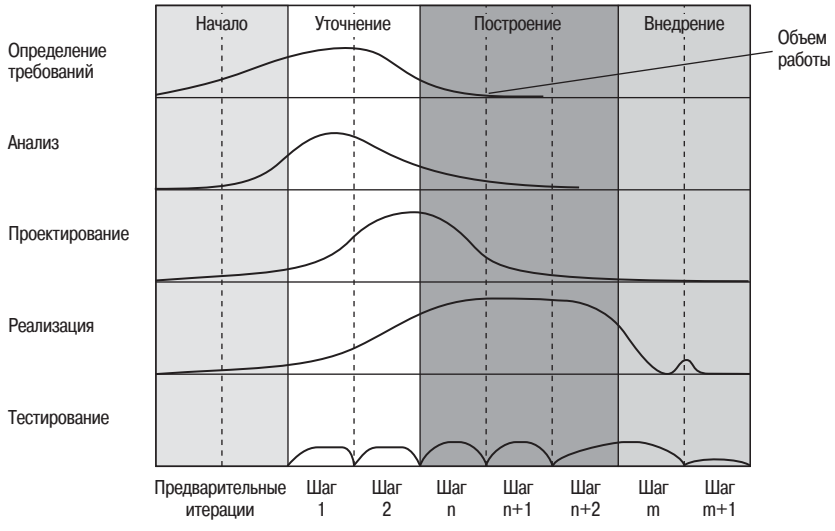


Рис. 2.7. *Относительный объем работы, выполняемый в каждом из пяти рабочих потоков по мере прохождения проекта по фазам. Адаптировано с рис. 1.5 [Jacobson 1] с разрешения издательства Addison-Wesley*

ный объем работы, выполняемый в каждом из пяти основных рабочих потоков по мере прохождения проекта по фазам.

> Объем работы, выполняемый в каждом из основных рабочих потоков, меняется в зависимости от фазы.

Как показано на рис. 2.7, в фазе Начало большая часть работы выпадает на определение требований и анализ. В фазе Уточнение основной акцент перемещается на требования, анализ и проектирование. Очевидно, что в фазе Построение основное внимание направлено на проектирование и реализацию. И наконец, в фазе Внедрение главными становятся реализация и тестирование.

> UP – процесс, ориентированный на цели в большей степени, чем на создание поставляемых артефактов.

Одной из замечательных особенностей UP является то, что он ориентирован на цели, а не на создание поставляемых артефактов. Каждая фаза завершается контрольной точкой, состоящей из набора условий, которым надо удовлетворить, и эти условия могут включать или не включать, в зависимости от конкретных потребностей проекта, создание отдельного, готового к поставке продукта.

Далее приводится краткий обзор каждой фазы UP.

2.9. Фазы UP

У каждой фазы есть цель, основная деятельность с акцентом на одном или более рабочих потоках, и контрольная точка. При обсуждении фаз мы будем придерживаться этих двух основных понятий.

2.9.1. Начало – цели

> Фаза Начало осуществляет инициирование проекта.

Цель фазы Начало – «сдвинуть проект с мертвой точки». Начало включает:

- Обоснование выполнимости – может включать разработку технического прототипа с целью проверки правильности технологических решений или концептуального прототипа для проверки бизнес-требований.
- Разработка экономического обоснования для демонстрации того, что проект обеспечит выраженную в количественном отношении коммерческую выгоду.
- Определение основных требований для создания предметной области системы.
- Выявление наиболее опасных рисков.

Основными исполнителями в данной фазе являются руководитель проекта и архитектор системы.

2.9.2. Начало – на что обращено внимание

В фазе Начало основное внимание обращено на определение требований и анализ. Однако если принято решение о создании технического или подтверждающего концепцию прототипа, может быть проведено некоторое проектирование и реализация. Тестирование обычно не применяется в данной фазе, поскольку единственными программными артефактами здесь являются прототипы, которые не будут больше нигде использоваться.

2.9.3. Начало – контрольная точка: Цели жизненного цикла

Тогда как многие SEP фокусируются на создании ключевых артефактов, UP применяет иной подход, ориентированный на цель. Каждая контрольная точка устанавливает определенные цели, которые должны быть выполнены для того, чтобы контрольная точка считалась пройденной. В частности, некоторые цели могут состоять в производстве определенных артефактов.

> Поставляемый артефакт создается, только если он действительно необходим в проекте.

Контрольной точкой фазы Начало являются Цели жизненного цикла. Условия, которые должны быть выполнены, чтобы эта контрольная точка была пройдена, приведены в табл. 2.1. Мы также предлагаем набор поставляемых артефактов, которые, возможно, потребуется создать для реализации этих условий. Однако следует запомнить, что поставляемый артефакт создается, если он действительно необходим в проекте.

Таблица 2.1

Условия принятия	Поставляемые артефакты
Заинтересованные стороны согласовали цели проекта.	Общее описание, определяющее основные требования, характеристики и ограничения проекта.
Заинтересованные стороны определили и одобрили предметную область системы.	Исходная модель прецедентов (выполненная только на 10–20%).
Заинтересованные стороны определили и одобрили ключевые требования.	Глоссарий проекта.
Заинтересованные стороны одобрили затраты и план работы.	Исходный план проекта.
Руководитель проекта сформировал экономическое обоснование проекта.	Экономическое обоснование.
Руководитель проекта провел оценку рисков.	Документ или база данных оценки рисков.
Посредством технических исследований и/или создания прототипа была подтверждена выполнимость.	Один или более одноразовых прототипов.
Архитектура намечена в общих чертах.	Документ с исходной архитектурой.

2.9.4. Уточнение – цели

Цели Уточнения можно описать следующим образом:

- создание исполняемой базовой версии архитектуры;
- детализация оценки рисков;
- определение атрибутов качества (скорости выявления дефектов, приемлемые плотности дефектов и т. д.);
- выявление прецедентов, составляющих до 80% от функциональных требований (что именно сюда входит, вы увидите в главах 3 и 4);
- создание подробного плана фазы Построение;
- формулировка предложения, включающего ресурсы, время, оборудование, штат и стоимость.

> В задачу фазы Уточнение входит создание неполной, но рабочей версии системы – исполняемой базовой версии архитектуры.

Основная цель фазы Уточнение – создание исполняемой базовой версии архитектуры. Это реальная исполняемая система, построенная соответственно заданной архитектуре. Это *не* прототип (который уйдет в корзину), а скорее всего, «первый срез» требуемой системы. Эта исполняемая базовая версия архитектуры будет дополняться по мере развития проекта и разовьется в окончательную поставляемую систему в фазах Построение и Внедрение. Поскольку следующие фазы основываются на результатах Уточнения, можно сказать, что Уточнение – решающая фаза. В книге этой фазе уделяется много внимания.

2.9.5. Уточнение – на что обращено внимание

В фазе Уточнение основное внимание в каждом из основных рабочих потоков обращено на следующее:

- определение требований – детализация предметной области системы и требований;
- анализ – выяснение, что необходимо построить;
- проектирование – создание стабильной архитектуры;
- реализация – построение базовой версии архитектуры;
- тестирование – тестирование базовой версии архитектуры.

Итак, основное внимание в фазе Уточнение направлено на рабочие потоки определения требований, анализа и проектирования. Реализация приобретает значение в конце фазы при создании исполняемой базовой версии архитектуры.

2.9.6. Уточнение – контрольная точка: Архитектура жизненного цикла

Контрольная точка – Архитектура жизненного цикла. Условия принятия контрольной точки перечислены в табл. 2.2.

Таблица 2.2

Условия принятия	**Поставляемые артефакты**
Создана гибкая надежная исполняемая базовая версия архитектуры.	Исполняемая базовая версия архитектуры.
Исполняемая базовая версия архитектуры демонстрирует, что важные риски были выявлены и учтены.	Статическая UML-модель. Динамическая UML-модель. UML-модель прецедентов.
Представление продукта стабилизировалось.	Общее описание проекта.
Оценка рисков пересмотрена.	Обновленная оценка рисков.

Таблица 2.2 (продолжение)

Условия принятия	Поставляемые артефакты
Экономическое обоснование проекта пересмотрено и одобрено всеми заинтересованными сторонами.	Обновленное экономическое обоснование проекта.
Создан достаточно детальный план проекта, что обеспечило возможность сформулировать реалистичную заявку на затраты времени, денег и ресурсов в следующих фазах. Заинтересованные стороны одобрили план проекта.	Обновленный план проекта.
Проведена проверка экономического обоснования проекта согласно плану проекта.	Экономическое обоснование проекта.
Заинтересованные стороны достигли соглашения о продолжении проекта.	Подписанный документ.

2.9.7. Построение – цели

> Построение превращает исполняемую базовую версию архитектуры в законченную рабочую систему.

Цель фазы Построение – завершить определение требований, анализ и проектирование и развить исполняемую базовую версию архитектуры, созданную в фазе Уточнение, в завершенную систему. Главная проблема Уточнения – *поддержание целостности архитектуры системы*. Очень часто при установлении сроков поставки и переходе к написанию кода начинается пренебрежение формальностями, что приводит к нарушению архитектурного представления, низкому качеству конечной системы и высоким затратам на обслуживание. Конечно, таких последствий следует избегать.

2.9.8. Построение – на что обращено внимание

Основное внимание в этой фазе уделено рабочему потоку реализации. В других рабочих потоках делается ровно столько, чтобы завершить определение требований, анализ и проектирование. Тестирование становится более важным: каждый новый инкремент надстраивается над предыдущим, поэтому здесь необходимо как тестирование отдельных элементов, так и совместное тестирование. Подводя итог, мы можем кратко охарактеризовать работу, выполняемую в каждом рабочем потоке, следующим образом:

- определение требований – выявление всех неучтенных требований;
- анализ – завершение аналитической модели;
- проектирование – завершение модели проектируемой системы;

- реализация – создание базовой функциональности;
- тестирование – тестирование базовой функциональности.

2.9.9. Построение – контрольная точка: Базовая функциональность

По существу, эта контрольная точка очень проста – программная система готова к бета-тестированию пользователем. Условия принятия данной контрольной точки приведены в табл. 2.3.

Таблица 2.3

Условия принятия	Поставляемые артефакты
Программный продукт достаточно стабилен и качественен для распространения среди пользователей.	Программный продукт. UML-модель. Тестовый комплект.
Заинтересованные стороны одобрили и готовы к введению программного продукта в свое окружение.	Руководство для пользователя. Описание данной версии.
Расхождения реальных расходов с предполагаемыми приемлемы.	План проекта.

2.9.10. Внедрение – цели

> Внедрение направлено на развертывание законченной системы в сообществе пользователей.

Внедрение начинается, когда завершено бета-тестирование и система окончательно развернута. Сюда относится устранение всех дефектов, обнаруженных при бета-тестировании, и подготовка к массовому выпуску программного обеспечения на все пользовательские сайты. Цели этой фазы можно обобщить следующим образом:

- исправление дефектов;
- подготовка пользовательских сайтов под новое программное обеспечение;
- настройка работоспособности программного обеспечения на пользовательских сайтах;
- изменение программного обеспечения в случае возникновения непредвиденных проблем;
- создание руководств для пользователей и другой документации;
- предоставление пользователям консультаций;
- проведение послепроектного анализа.

2.9.11. Внедрение – на что обращено внимание

Основное внимание концентрируется на рабочих потоках реализации и тестирования. Для исправления всех ошибок проектирования, выявленных при бета-тестировании, выполняется существенный объем проектирования. Надо стремиться к тому, чтобы в фазе Внедрение рабочие потоки определения требований и анализа оставались практически незадействованными. В противном случае с проектом не все в порядке.

- Определение требований – не проводится.
- Анализ – не проводится.
- Проектирование – изменение конструкции в случае выявления проблем при бета-тестировании.
- Реализация – настройка ПО под пользовательский сайт и исправление проблем, не выявленных при бета-тестировании.
- Тестирование – бета-тестирование и приемочные испытания на пользовательском сайте.

2.9.12. Внедрение – контрольная точка: Выпуск продукта

Это последняя контрольная точка: бета-тестирование, приемочные испытания и исправление дефектов завершены, продукт выпущен и принят в сообществе пользователей. Условия принятия этой контрольной точки приведены в табл. 2.4.

Таблица 2.4

Условия принятия	Поставляемые артефакты
Бета-тестирование завершено, необходимые изменения сделаны и пользователи согласны с тем, что система успешно развернута. Сообщество пользователей активно использует продукт.	Программный продукт.
Стратегии поддержки продукта согласованы с пользователями и реализованы.	План поддержки пользователя. Обновленные руководства для пользователей.

2.10. Что мы узнали

- Процесс производства программного обеспечения (SEP) превращает требования пользователя в ПО, определяя *кто что* делает и *когда*.
- Унифицированный процесс (UP) разрабатывается с 1967 года. Это сложившийся открытый SEP от авторов UML.
- Унифицированный процесс компании Rational (RUP) – это коммерческое расширение UP. Он полностью совместим с UP, но более полный и детализированный.

- UP (и RUP) должны настраиваться под каждый конкретный проект путем добавления внутренних стандартов и др.
- UP – это современный SEP, который является:
  - управляемым рисками и прецедентами (требованиями);
  - архитектуро-центричным;
  - итеративным и инкрементным.
- В UP программное обеспечение создается через итерации:
  - каждая итерация подобна мини-проекту, создающему часть системы;
  - для создания окончательной системы итерации надстраиваются друг над другом.
- В каждой итерации пять основных рабочих потоков:
  - определение требований – выяснение того, что должна делать система;
  - анализ – конкретизация и структурирование требований;
  - проектирование – реализация требований в архитектуре системы (как система это делает);
  - реализация – построение программного обеспечения;
  - тестирование – проверяется, работает ли должным образом реализация.
- В UP четыре фазы, каждая из которых заканчивается важной контрольной точкой:
  - Начало – проект сдвигается с «мертвой точки»: Цели жизненного цикла;
  - Уточнение – развитие архитектуры системы: Архитектура жизненного цикла;
  - Построение – построение программного обеспечения: Базовая функциональность;
  - Внедрение – развертывание программного обеспечения в пользовательской среде: Выпуск продукта.

II

Определение требований

3

Рабочий поток определения требований

3.1. План главы

В этой главе рассматривается все, что необходимо для понимания требований, предъявляемых к системе. Здесь вводится понятие требований и обсуждаются детали рабочего потока определения требований в UP. Также представлено расширение UP для работы с требованиями *без* применения прецедентов UML.

3.2. Рабочий поток определения требований

Как показано на рис. 3.2, большая часть работы в процессе определения требований выполняется в фазах Начало и Уточнение в самом начале жизненного цикла проекта. Это и не удивительно, поскольку невозможно продвигаться вперед, за фазу Уточнение, до тех пор, пока не известно, хотя бы приблизительно, что предполагается разрабатывать!

До начала работы над ОО анализом и проектированием уже необходимо иметь некоторое представление о том, что должно получиться в результате. Именно в этом состоит цель рабочего потока определения требований. С точки зрения ОО аналитика или дизайнера его цель – это поиск и достижение соглашения о функциях системы, написанного на языке пользователей системы. Создание высокоуровневой спецификации того, что должна делать система, – это часть процесса *выработки требований* (*requirements engineering*).

> Большая часть работы с требованиями проводится в начале проекта, в фазах Начало и Уточнение.

У любой заданной системы может быть множество различных заинтересованных сторон: разные типы пользователей, специалисты по эксплуатации, штат по обслуживанию, продаже, управлению и т. д. Выра-

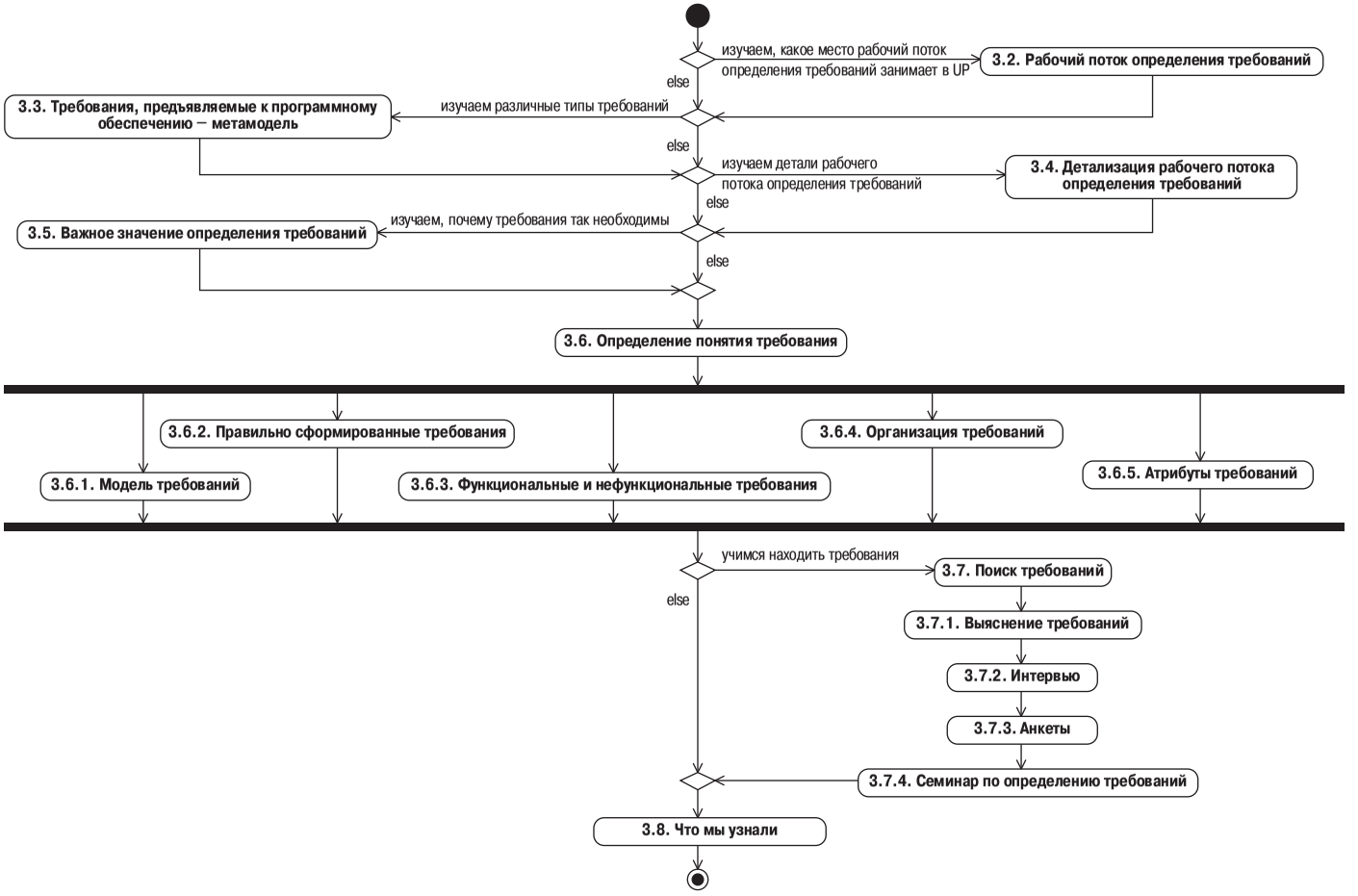


Рис. 3.1. *План главы*

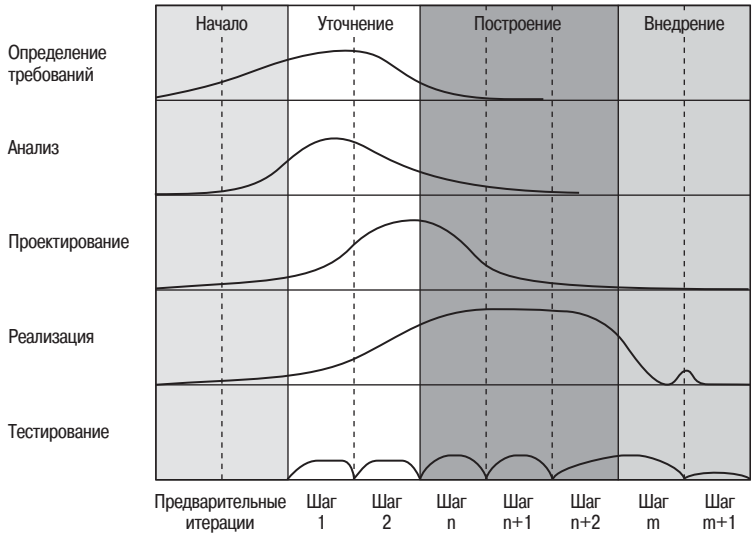


Рис. 3.2. *Определение требований выполняется в фазах Начало и Уточнение. Адаптировано с рис. 1.5 [Jacobson 1] с разрешения издательства Addison-Wesley*

ботка требований состоит в выявлении и классификации требований, предъявляемых к системе заинтересованными сторонами. Это переговорный процесс, поскольку часто выдвигаются противоречивые требования, которые должны быть согласованы. Например, одной группе хочется добавлять большое количество пользователей, что может привести к нереальному трафику в существующей базе данных и инфраструктуре связи. В настоящее время это наиболее распространенный конфликт, поскольку все большее и большее число компаний открывают части разработанных ими систем громадной аудитории пользователей через Интернет.

Некоторые книги по UML (и конечно, учебные курсы) отмечают, что прецеденты UML являются единственным способом фиксирования требований. Но это утверждение оказывается неверным при более тщательном рассмотрении. Прецеденты могут отражать только функциональные требования, которые являются описанием того, *что будет делать система*. Однако есть и другие требования, не относящиеся к функциональности, которые являются *описаниями ограничений, накладываемых на систему* (производительность, надежность и т. п.), и не могут быть представлены в виде прецедентов. Поэтому в книге предлагается надежный подход к выработке требований, иллюстрирующий дополнительные эффективные способы выявления *обоих* видов требований.

3.3. Метамодель требований, предъявляемых к программному обеспечению

На рис. 3.3 показана метамодель применяемого в данной книге подхода к выработке требований. Здесь используется синтаксис UML, который нами еще не был рассмотрен. Не беспокойтесь! Все это детально обсуждается позже, а пока необходимо знать лишь следующее:

- Пиктограммы, напоминающие папки, – это пакеты UML. Они являются механизмом группировки UML и содержат группы элементов модели UML. В сущности, они очень напоминают настоящие папки файловой системы тем, что используются для организации и группировки взаимосвязанных сущностей. Маленький треугольник в верхнем правом углу пакета указывает на то, что в пакете находится модель.
- Пиктограмма якоря показывает, что сущность, изображенная со стороны кружка, *содержит* сущность, находящуюся на другом конце линии.

Наша метамодель показывает, что Спецификация требований к программному обеспечению (Software requirements specification, SRS) включает Модель требований (Requirements model) и Модель прецедентов (Use case model). Эти две модели являются разными, но тем не менее взаимодополняющими способами представления требований, предъявляемых к системе.

Можно видеть, что в Модель требований входят Функциональные требования (требования, определяющие, что должна делать система) и Нефункцио-

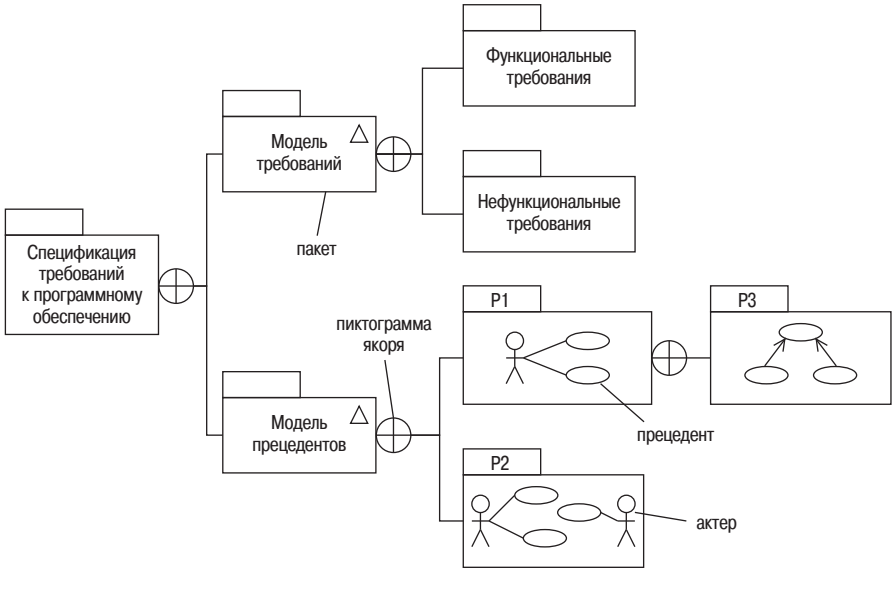


Рис. 3.3. *Метамодель требований*

нальные требования (требования, выражающие ограничения системы, не относящиеся к ее функциональности).

Модель прецедентов включает множество пакетов прецедентов (здесь показаны только три из них), которые содержат прецеденты (спецификации функциональных возможностей системы), актеров (внешние роли, непосредственно взаимодействующие с системой) и отношения.

По сути, SRS – это начальная стадия процесса построения программного обеспечения. Это отправная точка ОО анализа и проектирования.

Далее настоящая глава посвящена подробному рассмотрению разработки требований. Прецеденты и актеры обсуждаются в главе 4.

3.4. Детализация рабочего потока определения требований

На рис. 3.4 показаны конкретные задачи рабочего потока определения требований в UP. Такие диаграммы называют детализацией рабочего потока, поскольку они детализируют составляющие задачи определенного потока работ.

> Детализация рабочего потока показывает исполнителей и деятельности, вовлеченные в конкретный рабочий поток.

Детализации потока работ UP моделируются в виде исполнителей (пиктограммы в левой части рисунка) и деятельностей (пиктограммы в форме шестеренок). Разновидности UP, такие как RUP, могут ис-

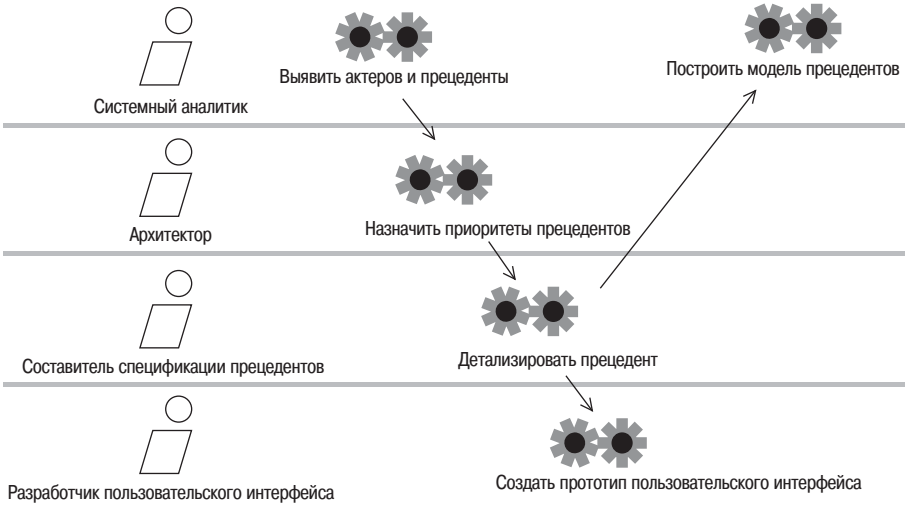


Рис. 3.4. *Задачи рабочего потока определения требований. Воспроизведено с рис. 7.10 [Jacobson 1] с разрешения издательства Addison-Wesley*

пользовать другие пиктограммы, но с той же семантикой (краткое обсуждение отношений между UP и RUP см. в разделе 2.4). Стрелки – это отношения, показывающие нормальный поток выполнения работы от одной задачи к следующей. Однако следует помнить, что это только приближенное представление рабочего потока для «усредненного» случая. Оно может и не быть точным представлением происходящего в действительности. В реальности в зависимости от обстоятельств некоторые задачи могут выполняться в другом порядке или параллельно.

Поскольку данная книга посвящена анализу и проектированию, основное внимание сосредоточено только на задачах, важных для ОО аналитиков и разработчиков. Поэтому нас интересует следующее:

- Выявление актеров и прецедентов.
- Детализация прецедента.
- Построение модели прецедентов.

Другие задачи рабочего потока определения требований не так важны для нас как аналитиков и разработчиков. Назначение приоритетов прецедентов – это деятельность в основном по планированию архитектуры и проекта, а Создание прототипа пользовательского интерфейса – это деятельность, касающаяся программирования. Более подробно о данных видах деятельности можно узнать в книге [Jacobson 1].

> Мы расширяем рабочий поток определения требований в UP, чтобы работать с требованиями, описанными на структурированном английском (или другом естественном) языке.

На рис. 3.4 показано, что стандартный рабочий поток UP сосредоточен на прецедентах, исключая все остальные методы выявления требований. В этом нет ничего страшного, но, как было отмечено выше, такой подход не может в достаточной мере реализовать нефункциональные требования к системе. Чтобы досконально рассмотреть все требования,

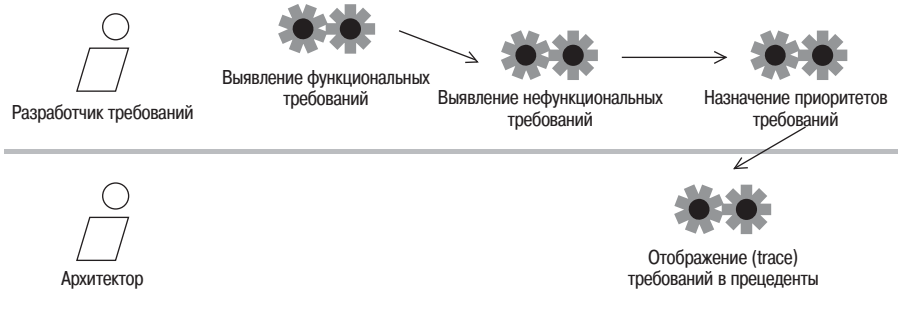


Рис. 3.5. *Расширенный рабочий поток определения требований: новые задачи и исполнители*

необходимо расширить рабочий поток определения требований в UP и добавить следующие новые задачи:

- Выявление функциональных требований.
- Выявление нефункциональных требований.
- Назначение приоритетов требований.
- Отображение (trace) требований в прецеденты.

Также был введен новый исполнитель – разработчик требований. Новые задачи и исполнители показаны на рис. 3.5.

3.5. Важное значение определения требований

Выработка требований – термин, используемый для описания деятельности по выявлению, документированию и обслуживанию ряда требований, предъявляемых к программной системе. Речь идет о выяснении того, что хотят получить от системы заинтересованные стороны.

Согласно [Standish 1], неполные требования и неучастие пользователей являлись двумя основными причинами провала проектов. Обе проблемы – результат ошибок в выработке требований.

Поскольку конечная программная система основывается на наборе требований, эффективная выработка требований – решающий фактор успеха в проектах по разработке программного обеспечения.

3.6. Определение понятия требования

Требование можно определить как «подробное описание того, что должно быть реализовано». Существует два основных типа требований:

- функциональные требования – какое поведение должна предлагать система;
- нефункциональные требования – особое свойство или ограничение, накладываемое на систему.

> Требования указывают, что должно быть построено, но не говорят, как это сделать.

Требования – это (по крайней мере, так должно быть) основа всех систем. Это, по сути, формулировка того, что должна делать система. В принципе требования должны быть *только* изложением того, *что* система должна делать, но не того, к*ак* она должна это делать. Это важное различие. Можно определить, *что* должна делать и какое поведение должна демонстрировать система, но при этом не обязательно что-либо говорить о том, *как* эта функциональность может быть реализована.

Хотя теоретически, конечно же, очень привлекательно отделить «что» от «как», но на практике набор требований скорее будет смесью «что» и «как». Отчасти из-за того, что обычно проще писать и понимать описание реализации, нежели абстрактное изложение проблемы. Отчасти потому, что могут существовать ограничения на реализацию, которые предопределяют «как» системы.

Несмотря на тот факт, что поведение системы, и особенно удовлетворение конечного пользователя, закладывается в процессе выработки требований, многие компании по-прежнему не считают это важным. Как было упомянуто, основная причина провалов проектов по разработке программного обеспечения заключается в проблемах с требованиями.

3.6.1. Модель требований

Во многих компаниях до сих пор нет формальной системы представления требований или модели требований. Программное обеспечение описывается в одном или более выполненных в произвольной форме и в разной степени полезных «документах с требованиями». Зачастую они написаны на естественном языке, имеют произвольные форму и размер. Для *любого* документа с требованиями, в какой бы форме он ни был представлен, ключевыми вопросами являются «насколько он полезен мне?» и «помогает ли он мне понять, что должна делать система?» К сожалению, полезность таких произвольным образом оформленных документов ограничена.

UP обладает формальным подходом к определению требований, основанным на модели прецедентов. Здесь мы расширяем его моделью требований, базирующейся на традиционных представлениях о функциональных и нефункциональных требованиях. Это расширение прямо соответствует более сложному подходу к выработке требований, применяемому в RUP. Наша метамодель требований (рис. 3.3) показывает, что SRS состоит из модели прецедентов *и* модели требований.

Модель прецедентов обычно создается в инструменте моделирования UML, таком как Rational Rose. Прецеденты подробно рассматриваются в главах 4 и 5.

Модель требований может быть создана в текстовом редакторе или в специальном инструментальном средстве выработки требований, например RequisitePro (*www.ibm.com*) или DOORS (*www.telelogic.com*). Мы рекомендуем использовать инструменты выработки требований по мере возможности. Как создавать правильно сформированные требования, обсуждается в следующих нескольких разделах.

3.6.2. Правильно сформированные требования

UML не дает каких-либо рекомендаций по написанию традиционных требований. Фактически требования в UML представляются исключительно через механизм прецедентов, который будет рассмотрен позже.

***Рис. 3.6.** Формат формулировки требований*

Однако многие разработчики моделей (включая и нас) полагают, что прецедентов недостаточно и все же нужны традиционные инструментальные средства выработки и управления требованиями.

> Для записи требований используйте утверждение «shall» («должен»).

Мы рекомендуем очень простой формат формулировки требований (рис. 3.6). Каждое требование имеет уникальный идентификатор (обычно это число), ключевое слово (shall (должен)) и описание действия. Преимущество применения унифицированной структуры состоит в упрощении задачи синтаксического разбора требований для инструментов управления требованиями, таких как DOORS.

3.6.3. Функциональные и нефункциональные требования

Полезно разделять требования на функциональные и нефункциональные. Существует много других способов систематизации требований, но для простоты начнем с этих двух категорий.

> Функциональное требование – это то, что система должна делать.

Функциональное требование – это формулировка того, что должна делать система, это описание назначения системы. Например, для банкомата (automated teller machine, ATM) можно было бы определить следующие функциональные требования:

1. Система ATM shall (должна) проверять действительность вставленной в банкомат карточки.
2. Система ATM shall (должна) проверять достоверность PIN-кода, введенного пользователем.
3. Система ATM shall (должна) выдавать по одной ATM-карточке не более $250 в сутки.

> Нефункциональное требование – ограничение, накладываемое на систему.

Нефункциональное требование – это ограничение, накладываемое на систему. Система ATM может иметь следующие нефункциональные требования?

1. Система ATM shall (должна) быть написана на C++.
2. Система ATM shall (должна) обмениваться информацией с банком, используя 256-разрядную кодировку.
3. Система ATM shall (должна) проверять действительность карточки ATM в течение не более трех секунд.
4. Система ATM shall (должна) проверять достоверность PIN-кода в течение не более трех секунд.

Можно заметить, что нефункциональные требования определяют или накладывают ограничения на то, как будет реализована система.

3.6.4. Организация требований

При использовании инструментального средства управления требованиями можно организовать требования в *таксономию* – иерархию типов требований, которая может использоваться при классификации требований. Типы требований применяются главным образом для создания небольших групп из громадного числа неструктурированных требований. Такими группами легче управлять, что делает работу с требованиями более эффективной.

Базовое разделение на функциональные и нефункциональные требования, описанное выше, является очень простой таксономией. Но можно пойти дальше и ввести категории требований, расширяя таксономию, как показано на рис. 3.7.

Рис. 3.7. *Иерархия типов требований*

Конкретные типы требований зависят от типа создаваемого программного обеспечения. Это особенно касается функциональных требований. Для нефункциональных требований набор типов требований, приведенный на рис. 3.7, довольно стандартный и неплох для начала.

Организация требований по типам полезна в случае большого числа требований (около сотни и более). Такой подход *особенно* хорош, если используется инструментальное средство выработки требований. Тогда для получения полезной информации можно будет делать запрос к модели требований по типу требований.

В принципе глубина иерархии типов требований может быть любой. На практике двух-трех уровней для системы средней сложности вполне достаточно.

3.6.5. Атрибуты требований

У каждого требования может быть ряд атрибутов, фиксирующих дополнительную информацию (метаданные) о требовании.

Каждый атрибут требования имеет описательное имя и значение. Например, у требования может быть атрибут dueDate (дата платежа), значением которого является дата выполнения данного требования. Также требование может иметь атрибут source (источник), в качестве значения которого выступает описание того, откуда возникло данное требование. Конкретный набор используемых атрибутов зависит от природы и нужд проекта и может меняться в зависимости от типа требования.

Наверное, самым распространенным атрибутом требований является priority (приоритет). Его значение определяет приоритет требования относительно остальных требований. Обычно для назначения приоритетов применяется набор критериев MoSCoW, описанный в табл. 3.1.

Таблица 3.1

Значения атрибута Priority	Семантика
Must have (обязан иметь)	Обязательные требования, являющиеся фундаментальными для системы.
Should have (должен иметь)	Важные требования, которые могут быть опущены.
Could have (мог бы иметь)	По-настоящему необязательные требования (реализуются, если есть на это время).
Want to have (хотел бы иметь)	Требования, которые могут подождать до следующих версий системы.

Если применяется схема MoSCoW, у каждого требования есть атрибут Priority, который может принимать одно из значений: M, S, C или W. Инструментальные средства выработки требований обычно обеспечивают возможность делать запрос к модели требований по значению атри-

бута. Таким образом, можно, например, сгенерировать список всех требований с максимальным приоритетом (Must have). Это очень полезно!

Преимущество набора критериев MoSCoW в его простоте. Однако в нем смешиваются два разных свойства требования: важность и очередность. Важность требования, единожды заданная, стремится оставаться относительно стабильной. А вот очередность, оцениваемая относительно остальных требований, в ходе проекта может меняться независимо от важности требования. Например, доступность ресурсов или зависимости от других требований.

RUP определяет более полный набор атрибутов требований, в котором разделены важность (Benefit) и очередность (TargetRelease). Атрибуты RUP представлены в табл. 3.2.

> Количество атрибутов требований должно быть минимальным, от этого проект только выиграет.

Выбор того, что применять – MoSCoW, RUP или какой-то другой набор атрибутов требований, – зависит от конкретного проекта. Главное при определении набора атрибутов – сделать его максимально простым. Необходимо выбирать только те атрибуты, которые полезны проекту. Если атрибут не приносит пользы, не надо его использовать.

Таблица 3.2

Атрибут	**Семантика**
Status (статус)	Может иметь одно из следующих значений: Proposed (предложенные) – требования, которые находятся в состоянии обсуждения и еще не утверждены. Approved (одобренные) – требования, утвержденные для реализации. Rejected (отклоненные) – требования, которые решено не реализовывать. Incorporated (включенные) – требования, которые были реализованы в определенной версии.
Benefit (полезность)	Может иметь одно из следующих значений: Critical (критичное) – требование *должно* быть реализовано, в противном случае система не будет принята заинтересованными сторонами. Important (важное) – требование может быть опущено, но это неблагоприятно отразится на удобстве использования системы и удовлетворении заинтересованных сторон. Useful (полезное) – требование может быть опущено без существенного влияния на приемлемость системы.
Effort (трудоемкость)	Оценка времени и ресурсов, необходимых для реализации возможности, выраженная в человеко-часах или другими методами, например методом функциональных точек (*www.ifpug.org*)

Атрибут	Семантика
Risk (риск)	Риск, связанный с добавлением этой возможности: High (высокий), Medium (средний) или Low (низкий).
Stability (стабильность)	Оценка вероятности того, что по каким-то причинам требование будет изменено: High (высокая), Medium (средняя) или Low (низкая).
TargetRelease (целевая версия)	Версия продукта, в которой требование должно быть реализовано.

3.7. Поиск требований

Требования следуют из контекста моделируемой системы. В этот контекст входят:

- непосредственные пользователи системы;
- другие заинтересованные стороны (например, руководители, специалисты обслуживания, установщики);
- другие системы, с которыми взаимодействует данная система;
- аппаратные устройства, с которыми взаимодействует данная система;
- правовые и регулирующие ограничения;
- технические ограничения;
- коммерческие цели.

Как правило, выработка требований начинается с документа, описывающего в общих чертах (vision) то, что собирается делать система и какие услуги она будет предоставлять ряду заинтересованных сторон. Назначение этого документа – обозначить наиболее важные цели системы с точки зрения заинтересованных сторон. Общее описание составляется системным аналитиком в UP-фазе Начало.

После общего описания системы начинается настоящая выработка требований. В следующих разделах мы обсудим некоторые методики выявления требований.

3.7.1. Выяснение требований — карта местности еще не территория

> Три фильтра – пропуск, искажение и обобщение – формируют естественный язык.

При выяснении у людей требований, предъявляемых к программной системе, вы всегда пытаетесь добиться от них точной картины, или карты, модели их сферы деятельности. Согласно книге Ноама Хомского (Noam Chomsky) «Syntactic Structures» [Chomsky 1], опубликованной в 1975 г. и посвященной трансформационной грамматике, подоб-

ная карта создается тремя процессами: пропуск (deletion), искажение (distortion) и обобщение (generalization). Эти процессы абсолютно необходимы, поскольку мы просто не располагаем механизмом познания, способным фиксировать каждый нюанс и деталь нашей сферы деятельности в некоторой воображаемой, подробной до мельчайших деталей карте. Поэтому приходится быть изобретательными. Мы осуществляем «выборку» из огромного массива возможной информации, применяя три фильтра:

- пропуск – информация отфильтровывается;
- искажение – информация изменяется взаимосвязанными механизмами вымысла и представления;
- обобщение – информация обобщается в правила, убеждения и понятия об истинности и ложности.

Эти фильтры образуют естественный язык. О них важно знать при тщательном сборе и анализе требований, поскольку для восстановления информации может понадобиться активно их идентифицировать и ставить под сомнение.

Ниже приведены примеры из системы управления библиотекой. Для каждого примера указаны вопрос, соответствующий данному фильтру, и возможный ответ:

- Пример: «Они используют систему для получения книг на время» – пропуск.

  Вопрос: Кто именно использует систему для получения книг на время?

  Ответ: читатели, другие библиотеки и библиотекари.

- Пример: «Тот, кто имеет книгу на руках, не может взять другую книгу до тех пор, пока не вернет предыдущую, срок возврата которой истек» – искажение.

  Вопрос: Существуют ли такие обстоятельства, при которых кто-либо мог бы взять новую книгу до того, как будут возвращены все имеющиеся на руках книги, срок возврата которых истек?

  Ответ: Фактически существует два обстоятельства, при которых право читателя на получение книг может быть восстановлено. Во-первых, все имеющиеся на руках книги, срок возврата которых истек, возвращены; во-вторых, за все невозвращенные книги, срок возврата которых истек, внесена плата.

- Пример: «Для получения книг у всех должен быть формуляр» – обобщение.

  Вопрос: Есть ли пользователи системы, которым не обязательно иметь формуляр?

  Ответ: Некоторые пользователи системы, например другие библиотеки, могут не иметь формуляра или имеют специальный формуляр с другими сроками и условиями возврата книг.

Два последних случая особенно интересны как примеры общеязыкового шаблона – квантора общности. Примеры кванторов общности:

- все
- каждый
- всегда
- никогда
- никто
- нисколько

При встрече с квантором общности всегда можно найти пропуск, искажение или обобщение. Кванторы общности обычно свидетельствуют о достижении границ или пределов ментальной карты субъекта. Обычно при проведении анализа следует ставить под сомнение кванторы общности. Чуть не написали «*всегда* следует ставить под сомнение кванторы общности», но тогда мы бы опровергали самих себя!

3.7.2. Интервью

Проведение интервью с заинтересованными сторонами является самым прямым способом сбора требований. Обычно более полную информацию можно получить при интервьюировании один на один. Основные моменты указаны ниже.

- Не заблуждайтесь по поводу решения – вам может *казаться*, что вы очень хорошо понимаете, чего хотят заинтересованные стороны. Но во время интервью это предположение не должно учитываться. Это единственная возможность выяснить, что им нужно на самом деле.
- Задавайте контекстно-свободные вопросы – вопросы, которые не предполагают какого-либо конкретного ответа и заставляют интервьюируемого говорить о проблеме. Например, вопрос «Кто использует систему?» является контекстно-свободным и способствует обсуждению, тогда как вопрос «Систему используете вы?» предполагает ответ «Да/нет» и заканчивает дискуссию.
- Слушать – единственный способ выяснить, чего хотят заинтересованные стороны, поэтому дайте им возможность поговорить. Позвольте им обсудить вопрос и рассмотреть его по-своему. Если вы ожидаете конкретных ответов на вопросы, у вас, возможно, сформировалось неверное решение и исходя из этого заблуждения вы задаете закрытые вопросы.
- Не занимайтесь телепатией. В сущности, мы все в некоторой степени телепаты. Телепатия – это заблуждение по поводу того, что вам известны чьи-то чувства, желания или мысли, базирующееся на том, что вы чувствовали бы, желали бы или думали бы в подобной ситуации. Это важная человеческая способность, потому что она является основой сочувствия. Однако это может внести субъективность в процесс выяснения требований, и все закончится тем, что вы хотите, а не в чем нуждаются *заинтересованные стороны*.
- Запаситесь терпением!

Обстановка, в которой проводится интервью, может оказывать большое влияние на качество получаемой информации. В частности, мы предпочитаем неформальную обстановку, например небольшое кафе, потому что здесь и интервьюер, и интервьюируемый могут расслабиться и разоткровенничаться.

Лучший способ записи информации во время интервью – бумага и ручка! Использование портативного компьютера отвлекает обе стороны и может напугать интервьюируемого. Мы предпочитаем графические диаграммы, выражающие идеи, как гибкий, не внушающий страха и графически богатый способ сбора информации. Более подробно об этом можно узнать по адресу *www.mind-map.com*.

После интервью информация анализируется и формируются некоторые предварительные требования.

3.7.3. Анкеты

Анкеты не заменяют интервью.

Если принято решение использовать анкеты *без* проведения интервью, можно оказаться в безвыходной ситуации, когда необходимо выбрать список вопросов до того, как стало известно, какие вопросы задавать.

Анкеты могут быть полезным дополнением к интервью. Они хорошо подходят для получения ответов на конкретные закрытые вопросы. Можно выделить из интервью ключевые вопросы, объединить их в анкету, а затем распространить ее на более широкую аудиторию. Это поможет проверить, правильно ли вы понимаете требования.

3.7.4. Семинар по определению требований

Семинар – это один из самых эффективных способов сбора информации, относящейся к требованиям. Для выявления ключевых требований организуется семинар и основные заинтересованные стороны приглашаются принять в нем участие.

Семинар должен сосредотачиваться на мозговой атаке. Это мощная техника сбора информации.

Во встрече должны принимать участие руководитель проекта, разработчик требований, основные заинтересованные стороны и специалисты в определенной области. Процедура проведения подобного семинара следующая:

1. Объясните, что сбор требований проводится методом мозговой атаки.
   1.1. Все идеи принимаются как хорошие.
   1.2. Идеи записываются, но *не* обсуждаются – никогда ни о чем не спорьте, просто записывайте и двигайтесь дальше. Все будет проанализировано позже.
2. Попросите членов команды назвать ключевые требования к системе.

2.1. Напишите каждое требование на стикере.

2.2. Приклейте стикер на стену или доску.

3. Затем можно еще раз просмотреть все выявленные требования и напротив каждого записать дополнительные атрибуты (см. раздел 3.6.5).

После встречи результаты анализируются и превращаются в требования, как обсуждалось ранее в этой главе. Результаты распространяются для сбора комментариев.

Выработка требований – это итеративный процесс, в котором требования выявляются по мере уточнения понимания нужд заинтересованных сторон. Возможно, придется организовать несколько семинаров по мере углубления вашего понимания.

Намного больше информации об организации семинаров и выработке требований можно найти в книгах [Leffingwell 1] и [Alexander 1].

3.8. Что мы узнали

В данной главе представлен рабочий поток определения требований в UP и общее обсуждение требований, предъявляемых к программному обеспечению. Вы узнали следующее.

- Большая часть работы в рабочем потоке определения требований выполняется в фазах Начало и Уточнение жизненного цикла проекта UP.
- Наша метамодель требований (рис. 3.3) показывает, что существует два способа представления требований: 1) функциональные и нефункциональные требования и 2) прецеденты и актеры.
- Детализация рабочего потока определения требований в UP включает следующие деятельности, представляющие интерес для ОО аналитиков и разработчиков:
  - Выявление актеров и прецедентов;
  - Детализация прецедента;
  - Построение модели прецедентов.
- Стандартный рабочий поток определения требований в UP расширяется:
  - актером: Разработчик требований;
  - деятельностью: Выявление функциональных требований;
  - деятельностью: Выявление нефункциональных требований;
  - деятельностью: Назначение приоритетов требований;
  - деятельностью: Отображение (trace) требований в прецеденты.
- Большинство провалов проектов обусловлено недостатками выработки требований.

- Существует два типа требований:
  - функциональные требования – какое поведение должна предлагать система;
  - нефункциональные требования – определенное свойство или ограничение, накладываемое на систему.
- Правильно сформированные требования должны быть выражены в простых структурированных фразах на английском языке, использующих выражения shall (должен), для того чтобы инструментальные средства выработки требований могли без труда проводить их синтаксический анализ.

  <id> <система> shall <действие>
- Модель требований содержит функциональные и нефункциональные требования к системе. Это может быть:
  - документ;
  - база данных в системе управления требованиями.
- Требования могут быть организованы в таксономию – иерархию типов требований, которая классифицирует требования.
- Требования могут иметь атрибуты – дополнительную информацию (метаданные), ассоциированную с каждым требованием:
  - например приоритет – MoSCoW (Must have, Should have, Could have, Want to have);
  - например атрибуты RUP (Status, Benefit, Effort, Risk, Stability, TargetRelease);
  - количество атрибутов требований должно быть минимальным, от этого проект только выиграет.
- Карта местности еще не территория. Естественный язык содержит:
  - пропуски – информация отфильтровывается;
  - искажения – информация модифицируется;
  - обобщения – информация обобщается в правила, убеждения и понятия об истинности и ложности.
- Кванторы общности (например, «все», «каждый») могут указывать границы чьей-либо ментальной карты сферы деятельности; их необходимо ставить под сомнение.
- Техники выявления требований:
  - интервью;
  - анкеты;
  - семинары.

4

Моделирование прецедентов

4.1. План главы

В этой главе обсуждаются основы моделирования прецедентов, что является еще одной формой выработки требований. Моделирование прецедентов будет рассмотрено так, как оно описано в UP. Основное внимание уделяется конкретным методам и стратегиям, которые могут применяться ОО аналитиком или дизайнером для эффективного моделирования прецедентов. В разделе приведены только самые простые прецеденты, чтобы ничто не отвлекало от рассмотрения этих методов. Полный (и более сложный) учебный пример можно найти на нашем веб-сайте *www.umlandtheunifiedprocess.com*.

UML не определяет какой-либо формальной структуры для описания прецедентов. В этом кроется проблема, поскольку разные разработчики моделей используют разные стандарты. Чтобы как-то справиться с этим, в этой главе и в учебном примере мы приняли простой и эффективный стандарт. Наш веб-сайт предоставляет схему XML (eXtensible Markup Language, расширяемый язык разметки) с открытым исходным кодом для прецедентов и актеров, которую читатели могут свободно использовать в своих проектах. Эти шаблоны основываются на лучших практиках индустрии и обеспечивают простой, но при этом эффективный стандарт записи спецификаций прецедентов.

На нашем веб-сайте также можно найти очень простую таблицу стилей XSL (eXtensible Stylesheet Language, расширяемый язык стилей), которая превращает XML-документы прецедентов в HTML, отображаемый в броузере. Эта таблица стилей – полезный пример. Она может быть запросто настроена под стандарты фирменного оформления или другие стандарты оформления документов для различных организаций. Подробное обсуждение XML выходит за рамки нашей книги, но для эффективной работы с такими документами, возможно, понадобится обратиться к учебникам по XML, таким как [Pitts 1] и [Kay 1].

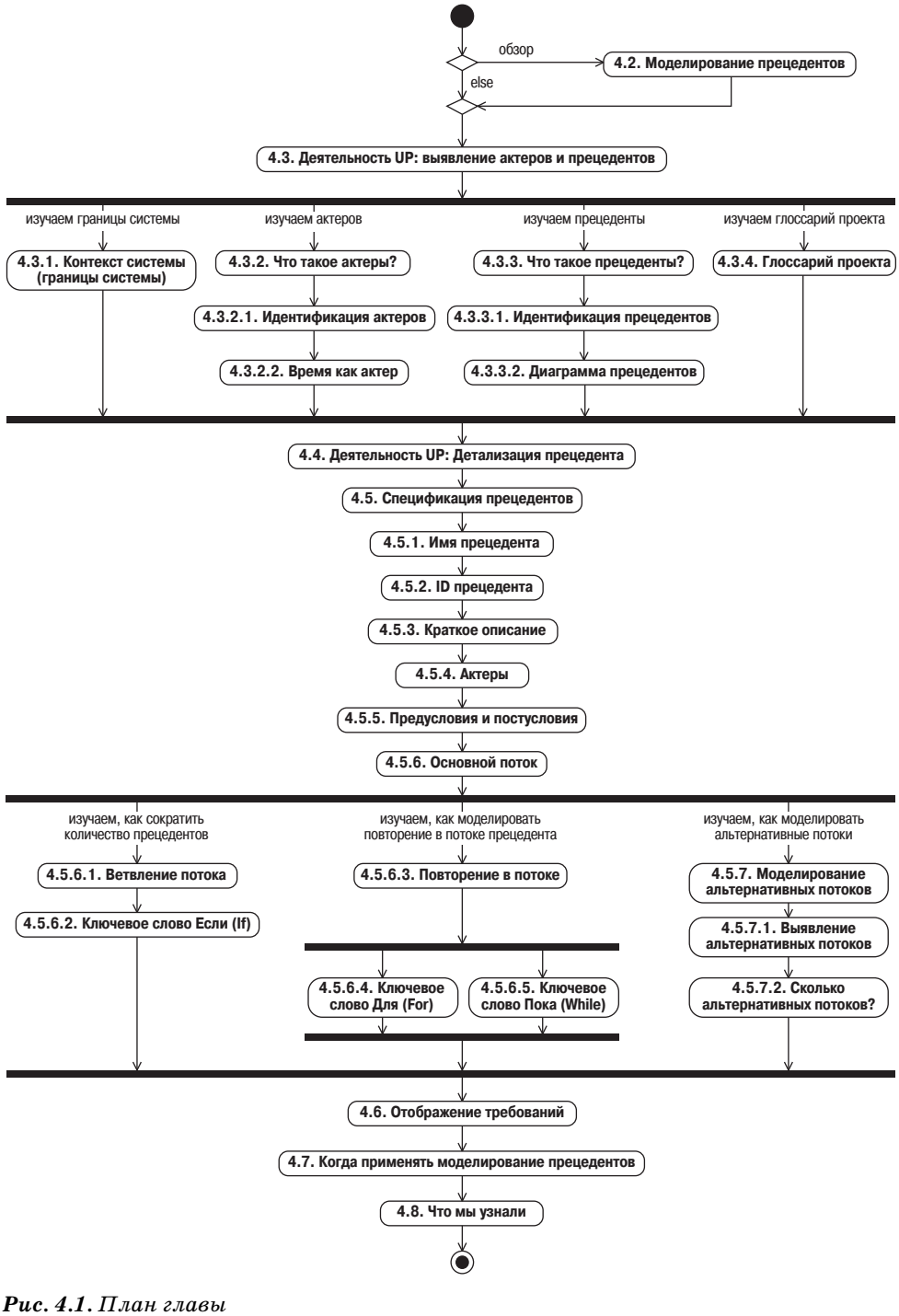


Рис. 4.1. *План главы*

Помимо схемы с открытым исходным кодом и таблиц стилей мы работаем над более гибким подходом под названием SUMR (Simple Use case Markup Restructured – простая реструктурированная разметка прецедентов, произносится «summer»). Это простой язык структурированной разметки текстов с открытым исходным кодом для прецедентов и актеров. Примеры схемы SUMR, синтаксические анализаторы и генераторы XML и HTML предоставлены на нашем веб-сайте, подробнее см. в разделе 2.2.

4.2. Моделирование прецедентов

Моделирование прецедентов – это форма выработки требований. В разделе 3.6 было показано, как создавать модель требований, объединяя функциональные и нефункциональные требования так называемым «традиционным» способом. Моделирование прецедентов – это другой, дополнительный способ выявления и документирования требований. Моделирование прецедентов обычно происходит следующим образом:

- Устанавливаются границы потенциальной системы.
- Выявляются актеры.
- Выявляются прецеденты:
  - определяется прецедент;
  - устанавливаются основные альтернативные потоки.
- Предыдущие шаги повторяются, пока прецеденты, актеры и границы системы не стабилизируются.

Обычно начинают с самой общей оценки границ системы, чтобы обозначить область моделирования. Затем все действия осуществляются итеративно и зачастую параллельно.

> Прецеденты – способ записи требований.

Результат этой деятельности – модель прецедентов. В этой модели четыре компонента:

- Граница системы – прямоугольник, очерчивающий прецеденты для обозначения края, или границы, моделируемой системы. В UML 2 эту границу называют *контекстом системы* (*subject*).
- Актеры – роли, выполняемые людьми или сущностями, использующими систему.
- Прецеденты – то, что актеры могут делать с системой.
- Отношения – значимые отношения между актерами и прецедентами.

Модель прецедентов является основным источником объектов и классов. Это основные исходные данные для моделирования классов.

4.3. Деятельность UP: Выявление актеров и прецедентов

Моделирование прецедентов включает выявление актеров и прецедентов.

В этом разделе основное внимание сосредоточено на деятельности Выявление актеров и прецедентов рабочего потока определения требований (см. раздел 3.4), изображенной на рис. 4.2. В разделе 4.4 мы рассмотрим деятельность Детализация прецедента.

Рассмотрим исходные данные для деятельности Выявление актеров и прецедентов.

- Бизнес-модель – может быть предоставлена в распоряжение разработчиков модели системы, но это не всегда выполняется. Если она есть, это превосходный источник для сбора требований.
- Модель требований – создание этой модели было описано в главе 3. Эти требования являются хорошим исходным материалом для процесса моделирования прецедентов. В частности, функциональные требования предложат прецеденты и актеров. Нефункциональные требования – то, о чем надо помнить при создании модели прецедентов.
- Список возможностей – это набор потенциальных требований, которые могут быть представлены в форме общего описания (vision) проекта или чего-либо подобного.

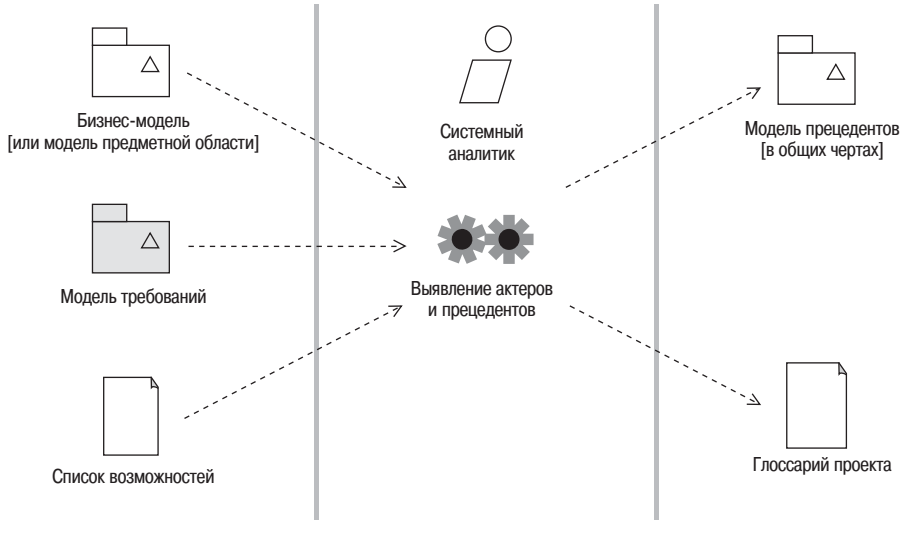


Рис. 4.2. *Деятельность UP: Выявление актеров и прецедентов. Адаптировано с рис. 7.11 [Jacobson 1] с разрешения издательства Addison-Wesley*

В оригинальной работе Джекобсона [Jacobson 1] вместо Модели требований (на рис. 4.2 этот прямоугольник затушеван, чтобы обозначить изменение, внесенное в исходный рисунок) присутствовали дополнительные требования. Сюда были включены требования (обычно нефункциональные), не относящиеся ни к одному из конкретных прецедентов. Этот документ являлся, главным образом, вместилищем для всех нефункциональных требований, противоречащих прецедентам. В нашем более устойчивом подходе к выработке требований дополнительные требования были разбиты на категории и включены в Модель требований в качестве подраздела.

4.3.1. Контекст системы (границы системы)

> Контекст системы отделяет систему от остального мира.

Первое, что необходимо сделать при построении системы, – обозначить ее границы. Иначе говоря, надо определить, что является *частью* системы (находится внутри границ системы) и что находится *вне* системы (вне ее границ). Это кажется очевидным, но встречается немало проектов, в которых из-за неясности границы системы возникают серьезные проблемы. Точное определение границ системы обычно играет важную роль в выявлении функциональных (а иногда и нефункциональных) требований. Мы уже были свидетелями того, как неполные и неправильно заданные требования могут стать основной причиной неудач проектов. В UML 2 границу системы называют *контекстом системы* (*subject*). Этого термина будем придерживаться и мы.

Контекст системы определяется тем, кто или что использует систему (т. е. актерами), и тем, какие конкретные преимущества система предлагает этим актерам (т. е. прецедентами).

Контекст изображается в виде прямоугольника с именем системы. Актеры размещаются *вне* границ блока, а прецеденты – *внутри*. В начале моделирования прецедентов имеется лишь предварительное представление о том, где находятся границы системы. По мере выявления актеров и прецедентов контекст системы обретает все более четкие очертания.

4.3.2. Что такое актеры?

> Актеры – это роли, исполняемые сущностями, непосредственно взаимодействующими с системой.

Актер определяет роль, которую выполняет некоторая внешняя сущность при *непосредственном* взаимодействии с данной системой. Он может представлять роль пользователя или роль, исполняемую другой

системой или частью аппаратных средств, которые касаются границ системы.

В UML 2 актеры могут также представлять другие контексты системы, обеспечивая возможность объединения разных моделей прецедентов.

> Роль подобна шляпе, которую надевают в определенной ситуации.

Для понимания актеров важно понимать концепцию ролей. Роль можно рассматривать как шляпу, которую надевают в определенной ситуации. Сущности могут играть несколько ролей одновременно либо исполнять их последовательно во времени. Это означает, что данная роль может исполняться многими разными сущностями одновременно либо последовательно во времени.

Например, если в системе определен актер Customer (покупатель), то эту роль могут исполнять реальные люди – Джим, Ила, Вольфганг, Роланд и многие другие. Эти люди могут играть и другие роли. Например, Роланд может быть и системным администратором (актер SystemAdministrator), и пользователем Customer.

Основная ошибка новичков в моделировании прецедентов – смешивание роли, выполняемой некоторой сущностью в контексте системы, с самой сущностью. Всегда спрашивайте себя: «Какую *роль* играет эта сущность по отношению к системе?» Так можно выявить общность поведения разных сущностей и таким образом упростить модель прецедентов.

В UML актеры изображаются так, как показано на рис. 4.3. Они могут быть изображены в виде пиктограммы класса с указанием стереотипа «actor» или в виде пиктограммы «анимационный человечек». Допускаются обе формы, но многие разработчики моделей предпочитают использовать «человечка» для тех ролей, которые, скорее всего, будут выполняться людьми, а пиктограммы класса для ролей, которые будут играть другие системы.

> Актеры являются внешними по отношению к системе.

Важно осознавать, что актеры всегда являются *внешними* по отношению к системе. Например, покупатель является внешним звеном в системе электронной торговли, такой как книжный интернет-магазин. Однако интересно отметить, что хотя сами актеры всегда находятся

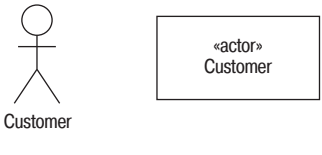


Рис. 4.3. *Варианты изображения актера*

вне системы, системы обычно имеют некоторое внутреннее представление одного или более актеров. Например, книжный интернет-магазин сохраняет сведения о большинстве покупателей, содержащие имя, адрес и другую информацию. Это внутреннее представление внешнего актера Customer. Важно четко разобраться в этом различии. Актер Customer является *внешним* по отношению к системе, но система может обслуживать класс CustomerDetails (информация о покупателе), который является *внутренним* представлением субъектов, играющих роль актера Customer.

4.3.2.1. Идентификация актеров

Для идентификации актеров необходимо ответить на вопросы: кто или что использует систему и какие роли выполняются актерами при взаимодействии с системой. Чтобы выявить роли, которые люди или сущности играют во взаимодействии с системой, можно учесть, а затем обобщить примеры конкретных людей и сущностей. Следующие вопросы помогут идентифицировать актеров.

> Чтобы выявить актеров, спросите: «Кто или что использует или взаимодействует с системой?»

- Кто или что использует систему?
- Какие роли они играю во взаимодействии?
- Кто устанавливает систему?
- Кто или что запускает и выключает систему?
- Кто обслуживает систему?
- Какие системы взаимодействуют с данной системой?
- Кто или что получает и предоставляет информацию системе?
- Происходит ли что-нибудь в точно установленное время?

При моделировании актеров необходимо помнить следующие моменты.

- Актеры всегда являются внешними по отношению к системе, следовательно, находятся вне вашего контроля.
- Актеры взаимодействуют *непосредственно* с системой – так они помогают в определении контекста системы.
- Актеры представляют роли, исполняемые людьми или сущностями по отношению к системе, а не конкретных людей или сущностей.
- Один человек или сущность может играть по отношению к системе множество ролей одновременно или последовательно во времени. Например, вы составляете и ведете учебные курсы. С точки зрения системы планирования курсов вы играете две роли: Trainer (инструктор) и CourseAuthor (автор курса).
- У каждого актера должно быть короткое, осмысленное с прикладной точки зрения имя.

- Каждого актера должно сопровождать краткое описание (одна или две строчки), объясняющее, что данный актер из себя представляет с прикладной точки зрения.
- Как и в классах, в обозначении актеров могут быть представлены атрибуты актера и события, которые он может получать. Такие обозначения не часто используются и редко отображаются на диаграммах прецедентов. Далее в этой книге они не упоминаются.

4.3.2.2. Время как актер

Если необходимо смоделировать что-то, происходящее с системой в определенный момент времени, но *не* инициируемое ни одним из актеров, можно ввести актера под названием Time (время) (рис. 4.4). В качестве примера приведем автоматическое сохранение резервной копии системы, выполняемое ежедневно вечером.

Рис. 4.4. *Актер Time*

4.3.3. Что такое прецеденты?

В книге «UML Reference Manual» [Rumbaugh 1] прецедент определен как «Описание последовательности действий, включая альтернативные и ошибочные последовательности, которые система, подсистема или класс могут осуществлять, взаимодействуя с внешними актерами».

> Прецедент описывает поведение, демонстрируемое системой с целью получения значимого результата для одного или более актеров.

Прецедент – это что-то, что должна делать система по желанию актера. Это «вариант использования» системы конкретным актером:

- прецеденты *всегда* инициируются актером;
- прецеденты *всегда* описываются с точки зрения актеров.

Обычно прецеденты рассматриваются на уровне системы, но согласно определению они могут применяться также и для описания «варианта использования» подсистемы (части системы) или даже отдельного класса. Прецеденты также могут быть очень эффективными при моделировании бизнес-процессов, хотя данный вопрос не рассматривается в этой книге.

На рис. 4.5 показана пиктограмма UML для прецедентов. Имя прецедента может быть написано внутри овала или под ним.

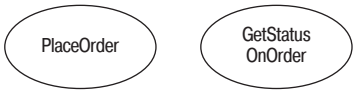


Рис. 4.5. *Пиктограмма прецедента*

4.3.3.1. Идентификация прецедентов

> Чтобы найти прецедент, надо спросить: «Как каждый из актеров использует систему?» и «Что система делает для каждого актера?»

Идентификацию прецедентов лучше всего начать со списка актеров, а затем рассмотреть, как каждый актер собирается использовать систему. С помощью такой стратегии можно получить список потенциальных прецедентов. Каждому прецеденту должно быть присвоено короткое описательное имя, представляющее собой глагольную группу (в конце концов, прецедент означает «*выполнить*» что-нибудь!).

Во время идентификации прецедентов могут обнаружиться некоторые новые актеры. Это нормально. Иногда приходится очень тщательно анализировать функциональность системы, чтобы выявить всех актеров, причем *правильно* выявить.

Моделирование прецедентов – итеративный процесс, осуществляемый путем поэтапного уточнения. Все начинается с имени прецедента, а детали дополняются позже. Эти детали состоят из исходного краткого описания, которое уточняется до полной спецификации. Ниже приводится полезный список вопросов, которые можно задавать при идентификации прецедентов.

- Какие функциональные возможности понадобятся конкретному актеру от системы?
- Система сохраняет и извлекает информацию? Если да, какой из актеров инициирует это поведение?
- Что происходит, когда система изменяет состояние (например, при запуске и выключении системы)? Кто-нибудь из актеров получает при этом уведомление?
- Какие-либо внешние события оказывают влияние на систему? Как система узнает об этих событиях?
- Система взаимодействует с какой-либо внешней системой?
- Система генерирует какие-либо отчеты?

4.3.3.2. Диаграмма прецедентов

На диаграмме прецедентов контекст модели прецедентов изображается в виде блока с именем контекста. Этот блок является контекстом и, как упоминалось в разделе 4.3.1, он представляет границу системы,

моделируемую прецедентами. Актеры располагаются вне контекста (они внешние по отношению к системе), а прецеденты, составляющие поведение системы, располагаются внутри контекста (они внутренние по отношению к системе). Это проиллюстрировано на рис. 4.6.

Отношение между актером и прецедентом обозначается сплошной линией. Это символ ассоциации в UML. Ассоциациям посвящена глава 9. Ассоциация между актером и прецедентом показывает, что актер и прецедент каким-то образом взаимодействуют.

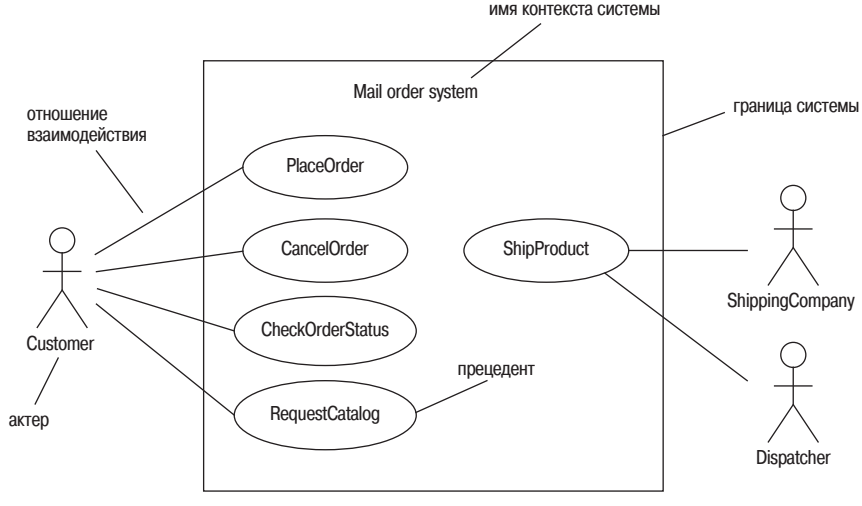


Рис. 4.6. *Диаграмма прецедентов*

4.3.4. Глоссарий проекта

> В глоссарии проекта должна быть отражена бизнес-терминология и жаргон.

Глоссарий проекта – возможно, один из самых важных артефактов проекта. Любая область деятельности имеет собственный уникальный язык, и основной целью процесса выработки и анализа требований является понимание и фиксация этого языка. Глоссарий обеспечивает словарь ключевых деловых терминов и определений. Он должен быть понятен всем участникам проекта, включая все заинтересованные стороны.

Кроме определения ключевых терминов глоссарий проекта должен включать синонимы и омонимы.

- Синонимы – это разные слова, обозначающие одно и то же. ОО аналитик должен выбрать и придерживаться одного из этих слов (того, которое имеет наиболее широкое применение). Остальные вариан-

ты должны быть полностью исключены из моделей. Это обусловлено тем, что разрешение на применение синонимов может привести к возникновению двух более или менее одинаковых классов с разными именами. Кроме того, если позволить эпизодическое употребление всех синонимов, можно с уверенностью ожидать, что семантика терминов со временем будет отличаться.

- Омонимы – это слова, одинаковые по звучанию, но разные по значению. В этом случае всегда возникают проблемы общения, поскольку разные стороны говорят на разных языках, но при этом *уверены*, что говорят об одном и том же. Способ решения – выбрать одно значение для определенного термина, а для других омонимов, возможно, ввести новые термины.

В глоссарии проекта должен быть указан предпочтительный термин и под его определением перечислены все синонимы. При этом, вероятно, некоторым деловым партнерам придется привыкать к другой терминологии. Обычно тяжело заставить заинтересованные стороны изменить используемый ими язык, но при некоторой настойчивости этого удается добиться.

UML не устанавливает каких-либо стандартов для глоссария проекта. Лучше всего, если глоссарий будет максимально простым и кратким. Можно применять формат словаря с сортировкой слов и выражений по алфавиту. Возможно, будет достаточно простого текстового документа, но для больших проектов может потребоваться сетевой глоссарий в формате HTML/XML или даже в виде несложной базы данных. Запомните, что чем глоссарий понятней и проще в использовании, тем сильнее его положительное влияние на проект.

Часть примера глоссария проекта приведена в табл. 4.1. Если у слова нет синонимов или омонимов, это явно указывается словом «нет» (поле не оставляется пустым). Такой прием подчеркивает, что данный вопрос был рассмотрен. Это придает глоссарию хороший стиль.

Термины и определения глоссария проекта также будут использоваться в модели UML. Необходимо гарантировать, что эти два документа синхронизированы. К сожалению, большинство инструментальных средств моделирования UML не обеспечивают какой-либо поддержки синхронизации модели UML и глоссария, поэтому подобная синхронизация обычно выполняется вручную.

Таблица 4.1. Глоссарий проекта для Clear View Training ECP (платформа для электронной торговли)

Термин	Определение
Catalog	Список всех продуктов, имеющихся в продаже в Clear View Training в настоящее время. Синонимы: нет Омонимы: нет

Таблица 4.1 (продолжение)

Термин	Определение
Checkout	Электронный аналог реальной кассы супермаркета. Место, где покупатели могут оплатить продукты, находящиеся в их корзине для покупок. Синонимы: нет Омонимы: нет
Clear View Training	Компания с ограниченной ответственностью, специализирующаяся на продаже книг и CD. Синонимы: CVT Омонимы: нет
Credit card	Карточка, например VISA или Mastercard, которая может использоваться для оплаты. Синонимы: карточка Омонимы: нет
Customer	Сторона, покупающая продукты или сервисы у Clear View Training. Синонимы: нет Омонимы: нет

4.4. Деятельность UP: Детализация прецедента

После создания диаграммы прецедентов и выявления актеров и ключевых прецедентов приступают к точному определению каждого прецедента по очереди. Эту деятельность UP известна как Детализация прецедента (рис. 4.7).

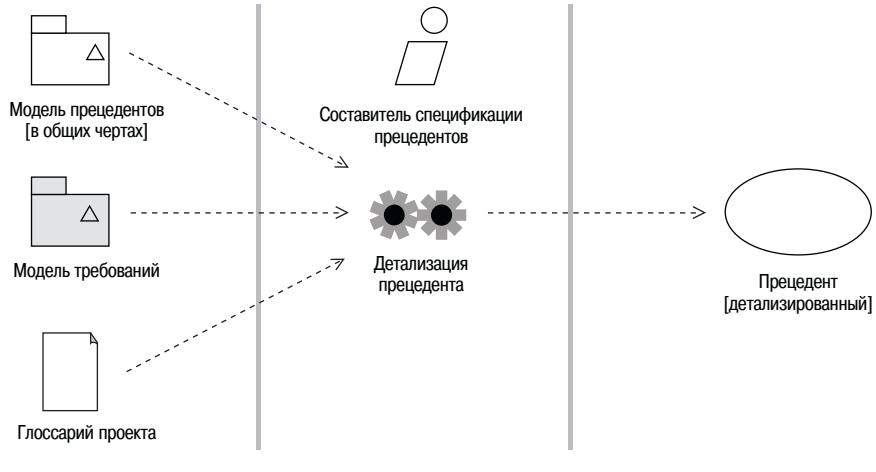


Рис. 4.7. *Деятельность UP: Детализация прецедента. Адаптировано с рис. 7.1 [Jacobson 1] с разрешения издательства Addison-Wesley*

Здесь важно отметить, что некоторые или все прецеденты можно описывать по мере их выявления, не соблюдая строгой очередности. Напротив, в книге всегда очень сложно представить параллельную деятельность, поскольку книга по природе своей линейна!

Итог этой деятельности – более детализированный прецедент. Сюда входят, по крайней мере, имя прецедента и его спецификация.

4.5. Спецификация прецедентов

UML-стандарта для спецификации прецедента не существует. Однако широко используется шаблон, приведенный на рис. 4.8. Есть и более сложные шаблоны, но из собственного опыта мы знаем, что при моделировании прецедентов лучше всего придерживаться максимальной простоты.

> Выберите стандарт для спецификации прецедентов.

Важно, чтобы организация выбрала стандарт для спецификаций прецедентов, который будет постоянно использоваться в проектах. Отсутствие таких стандартов в компании делает весь процесс моделирования прецедентов излишне сложным. В рамках одного проекта возникает масса разных форматов, уровней детализации или даже интерпретаций того, что является, а что не является прецедентом. Простой и эффективный стандарт для спецификаций прецедентов может помочь обеспечить ус-

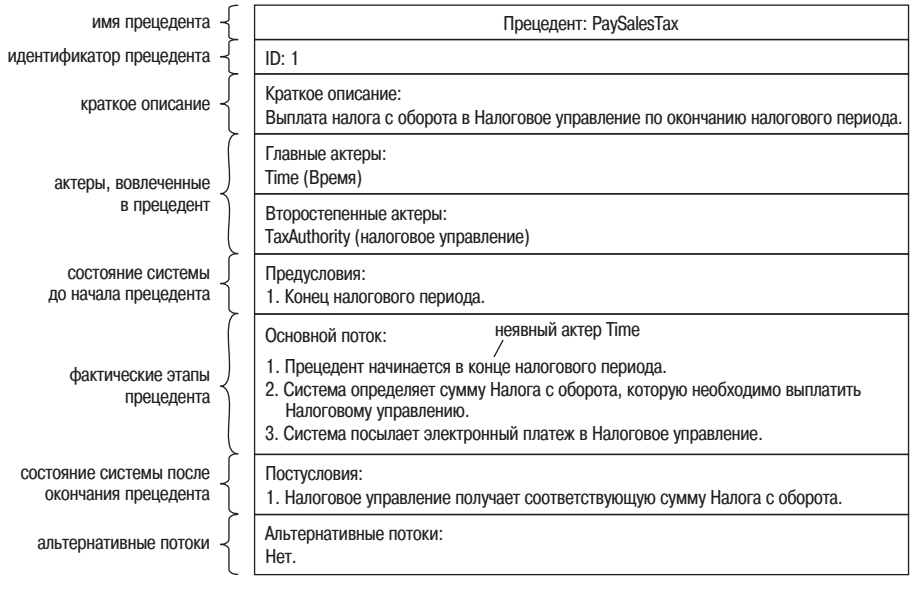


Рис. 4.8. *Шаблон спецификации прецедента*

пешный анализ прецедентов. В этой и следующей главах мы предлагаем такой стандарт.

В наш шаблон простой спецификации прецедента входит следующая информация:

- имя прецедента;
- ID прецедента;
- краткое описание – абзац, в котором изложена цель прецедента;
- актеры, задействованные в прецеденте;
- предусловия – условия, которые должны выполниться, чтобы прецедент мог осуществиться; это ограничения на состояние системы;
- основной поток – шаги выполнения прецедента;
- постусловия – условия, которые должны выполниться по окончанию прецедента;
- альтернативные потоки – список альтернативных основному потоку событий.

Позже, когда будут рассматриваться более сложные прецеденты, этот шаблон будет расширен для включения дополнительной информации.

Прецедент, представленный на рис. 4.8, касается выплаты Налога с оборота – формы налога, взимаемого с продаж во многих странах. В данном примере Налоговое управление в любом случае тем или иным образом получает соответствующую сумму налога. Поэтому факт получения денег здесь заявлен как постусловие прецедента.

> Записывайте прецеденты на структурированном естественном языке.

Хороший способ записи прецедента – «структурированный английский» (или русский, или любой другой язык, являющийся для вас родным). В следующих разделах будет представлен простой стиль, который можно использовать для эффективного выражения прецедентов.

4.5.1. Имя прецедента

Не существует UML-стандарта по присваиванию имен прецедентам. Мы всегда записываем имена прецедентов в стиле UpperCamelCase: отдельные слова имени прецедента записываются слитно, каждое слово начинается с заглавной буквы.

Прецеденты описывают поведение системы, поэтому имя прецедента всегда должно быть глаголом или глагольной группой, например PaySalesTax (выплата налога с оборота). Всегда надо стремиться выбрать имя, короткое и описательное одновременно. Человек, работающий с моделью прецедентов, должен по одному его имени четко понимать назначение моделируемой бизнес-функции или процесса. В этой и следующих главах можно найти множество примеров имен прецедентов.

Имя прецедента является его уникальным идентификатором в рамках модели прецедентов.

4.5.2. ID прецедента

Хотя имена прецедентов в рамках одной модели прецедентов должны быть уникальными, со временем они могут меняться. Следовательно, надо ввести другой постоянный идентификатор, уникально идентифицирующий конкретный прецедент в проекте. Обычно мы используем просто число.

При работе с альтернативными потоками (раздел 4.5.7) можно применять иерархическую систему нумерации. В этом случае легко устанавливается взаимосвязь между альтернативным и основным потоками. Например, если прецедент стоит под номером X, его альтернативные потоки будут пронумерованы как X.1, X.2,..., X.n.

4.5.3. Краткое описание

Это должен быть один абзац, в котором изложена цель прецедента. Попытайтесь уловить суть прецедента – прикладное значение, которое он имеет для актеров.

4.5.4. Актеры

> Главные актеры инициируют прецедент.

С точки зрения отдельного прецедента существует два типа актеров:

- главные актеры – актеры, инициирующие прецедент;
- второстепенные актеры – актеры, взаимодействующие с прецедентом после его инициации.

> Второстепенные актеры не инициируют прецедент.

Каждый прецедент всегда инициируется одним актером. Однако один и тот же прецедент в разные моменты времени может инициироваться разными актерами. Любой актер, который может инициировать прецедент, является главным актером. Все остальные актеры – второстепенные.

4.5.5. Предусловия и постусловия

Предусловия и постусловия – это ограничения.

- Предусловия ограничивают состояние системы, необходимое для запуска прецедента. Они как привратники, которые не дают актеру инициировать прецедент до тех пор, пока не будут выполнены все их условия.

- Постусловия ограничивают состояние системы после выполнения прецедента.

> Предусловия ограничивают состояние системы, необходимое для запуска прецедента. Постусловия ограничивают состояние системы после выполнения прецедента.

На это можно посмотреть по-другому. Предусловия определяют условия, которые должны быть истинными *для того*, чтобы прецедент мог быть инициирован. Постусловия определяют, какие условия будут истинными *после* выполнения прецедента. Предусловия и постусловия помогают спроектировать правильно функционирующую систему.

Предусловия и постусловия должны быть простыми выражениями о состоянии системы, которые затем определяются как истинные или ложные. Их называют логическими условиями.

Если прецедент не имеет предусловий или постусловий, хорошим стилем считается надпись «Нет» в соответствующих разделах спецификации прецедента. Это показывает, что данный вопрос был рассмотрен, иначе незаполненный раздел указывает на некоторую двусмысленность.

4.5.6. Основной поток

> Основной поток описывает «идеальный» ход развития событий в прецеденте.

Этапы прецедента представляются в виде потока событий. Прецедент можно представить как дельту реку с множеством ответвляющихся рукавов. У каждого прецедента есть один основной поток (основной рукав дельты). Остальные, меньшие рукава – это альтернативные потоки прецедента. Эти альтернативные потоки могут перехватывать ошибки, ответвления и прерывания основного потока. Основной поток иногда называют *основным сценарием* (*primary scenario*), а альтернативные потоки – *второстепенными сценариями* (*secondary scenarios*).

Основной поток регистрирует этапы прецедента, отражающие «идеальную» ситуацию, когда все идет, как ожидается и хочется, то есть не возникает ошибок, отклонений, прерываний или ответвлений.

Отклонения от основного потока можно смоделировать двумя способами, которые вскоре будут рассмотрены.

1. Простые отклонения – создаются ветвления основного потока (раздел 4.5.6.1).
2. Сложные отклонения – создаются альтернативные потоки (раздел 4.5.7).

Основной поток *всегда* начинается с действий главного актера, направленных на инициацию прецедента. Удачным способом начала потока можно считать следующую форму записи:

1. Прецедент начинается, когда <актер> <действие>.

Помните, что время тоже может быть актером, поэтому прецедент может начинаться временным выражением, как на рис. 4.8.

Поток событий состоит из последовательности коротких этапов, декларативных, пронумерованных и упорядоченных во времени. Каждый этап потока прецедента должен быть выражен в следующей форме:

<номер> <кто-либо> <совершает некоторое действие>.

Поток событий прецедента может быть представлен в повествовательной форме, однако это не рекомендуется, поскольку данная форма является слишком неопределенной.

Ниже приведен пример нескольких этапов прецедента PlaceOrder (разместить заказ).

1. Прецедент запускается, когда покупатель выбирает опцию «разместить заказ».
2. Покупатель заполняет в форме свои имя и адрес.

Это правильно сформированные прецеденты. В обоих случаях мы имеем простое декларативное выражение о том, что некоторая сущность осуществляет некоторое действие. А вот пример неверного описания этапа прецедента:

2. Вводятся данные покупателя.

Использование пассивного залога для описания любого этапа является неверным. На этом конкретном этапе фактически допущены три важных пропуска.

- Кто вводит данные покупателя?
- Куда вводятся эти данные?
- Что конкретно подразумевается под «данными покупателя»?

Важно различать и избегать пропусков при написании потоков прецедентов, даже если об этом можно сказать или догадаться исходя из контекста. Прецедент должен быть точным описанием части выполняемых функций системы!

Если в процессе анализа встречаются неопределенности, пропуски или обобщения, полезно ставить следующие вопросы.

- Кто именно...?
- Что именно...?
- Когда именно ...?
- Где именно ...?

4.5.6.1. Ветвление потока

Спецификация UML не определяет способа представления ветвления потока.

Мы пользуемся идиомой, которая позволяет представить ветвление простым способом без записи отдельного альтернативного потока. Для этого используется ключевое слово Если (If).

> Ветвление потока можно сократить, уменьшая число прецедентов, но пользоваться этим надо умеренно!

Стоит отметить, что некоторые разработчики моделей прецедентов выступают против ветвления в прецедентах. Они утверждают, что если есть ответвление, значит, должен быть описан новый альтернативный поток. Строго говоря, этот аргумент заслуживает внимания. Однако мы занимаем более пракматичную позицию и считаем, что небольшое простое ветвление потока допустимо, потому что оно сокращает общее число альтернативных потоков и обеспечивает более компактное представление требований.

4.5.6.2. Ключевое слово Если (If)

Для представления ответвления потока используйте ключевое слово Если (If). Пример, приведенный на рис. 4.9, показывает хорошо структурированный поток событий с двумя ветвями. Каждая ветвь начинается

Прецедент: ManageBasket
ID: 2
Краткое описание: Покупатель меняет количество товаров в корзине.
Главные актеры: Покупатель
Второстепенные актеры: Нет.
Предусловия: 1. Содержимое корзины для покупок является видимым.
Основной поток: 1. Прецедент начинается, когда Покупатель выбирает товарную позицию в корзине. 2. Если Покупатель выбирает «удалить позицию». 2.1. Система удаляет позицию из корзины. 3. Если Покупатель вводит новое количество. 3.1. Система обновляет количество товаров в корзине.
Постусловия: Нет.
Альтернативные потоки: Нет.

Рис. 4.9. *Прецедент с двумя ветвлениями*

с ключевого слова Если и простого логического выражения, такого как Если пользователь вводит новое количество, которое может быть истинным (true) или ложным (false). Структурированный текст под выражением Если – это то, что произойдет, если логическое выражение истинно. С помощью отступов и нумерации можно четко обозначить тело выражения Если *без* использования слов конец если (endif) или другого завершающего выражение синтаксиса.

Поскольку события ветвления могут произойти, а могут и не произойти в зависимости от обстоятельств, они не могут генерировать постусловия, которые *должны* выполняться обязательно. Поэтому ветвление *может* сократить число постусловий прецедента.

4.5.6.3. Повторение в потоке

Иногда некоторое действие в потоке событий необходимо повторить несколько раз. В моделировании прецедентов это встречается не часто, но на всякий случай полезно иметь некоторую стратегию для обработки таких вариантов.

Спецификация UML не определяет способа представления повторений в потоке, поэтому мы предлагаем простые выражения с ключевыми словами Для (For) и Пока (While).

4.5.6.4. Ключевое слово Для (For)

Смоделировать повторение можно с помощью ключевого слова Для. Формат следующий:

```
n. Для (выражение, описывающее итерации)
  n.1. Сделать что-то
  n.2. Сделать что-то другое
  n.3. …
n+1.
```

Выражение, описывающее итерации, – это некоторое выражение, результат которого – количество итераций. Каждая структурированная строка после выражения Для повторяется столько раз, сколько определено в выражении. Пример приведен на рис. 4.10.

4.5.6.5. Ключевое слово Пока (While)

Ключевое слово Пока (While) используется для моделирования последовательности действий в потоке событий, которые осуществляются до тех пор, пока некоторое логическое условие истинно. Формат в данном случае следующий:

```
n. Пока (логическое условие)
  n.1. Сделать что-то
  n.2. Сделать что-то другое
  n.3. …
n+1.
```

Прецедент: FindProduct
ID: 3
Краткое описание: Система ищет некоторые продукты на основании критерия поиска, заданного Покупателем, и выводит их на экран для Покупателя.
Главные актеры: Покупатель
Второстепенные актеры: Нет.
Предусловия: Нет.
Основной поток: 1. Прецедент начинается, когда Покупатель выбирает опцию «найти продукт». 2. Система запрашивает у Покупателя критерий поиска. 3. Покупатель вводит запрашиваемый критерий. 4. Система ищет продукты, соответствующие критерию Покупателя. 5. Если система находит соответствующие продукты, тогда 5.1. Для каждого найденного продукта 5.1.1. Система выводит на экран миниатюрное представление продукта. 5.1.2. Система выводит на экран краткое описание продукта. 5.1.3. Система выводит на экран цену продукта. 6. Иначе (Else) 6.1. Система сообщает Покупателю о том, что соответствующие продукты не найдены.
Постусловия: Нет.
Альтернативные потоки: Нет.

Рис. 4.10. *Моделирование повторений с помощью ключевого слова «Для»*

Как и ключевое слово Для, Пока используется не часто. Пример приведен на рис. 4.11. Последовательность структурированных строк после выражения Пока повторяется до тех пор, пока логическое условие, определенное в блоке Пока, не станет ложным.

4.5.7. Моделирование альтернативных потоков

> У каждого прецедента есть один основной поток и может быть множество альтернативных потоков.

У каждого прецедента есть основной поток и может быть множество альтернативных потоков. Они являются альтернативными путями в прецеденте, которые перехватывают ошибки, ответвления и прерывания основного потока. Как мы видели, спецификация прецедента включает основной поток и список имен альтернативных потоков.

> Альтернативные потоки часто не возвращаются в основной поток прецедента.

Прецедент: ShowCompanyDetails
ID: 4
Краткое описание: Система выводит данные о компании для Покупателя.
Главные актеры: Покупатель
Второстепенные актеры: Нет.
Предусловия: Нет.
Основной поток: 1. Прецедент начинается, когда Покупатель выбирает опцию «показать данные о компании». 2. Система выводит на экран веб-страницу с данными о компании. 3. Пока Покупатель просматривает данные о компании. 3.1. Система воспроизводит некоторую фоновую мелодию. 3.2. Система отображает специальные предложения в баннере.
Постусловия: 1. Система показала данные о компании. 2. Система воспроизвела фоновую мелодию. 3. Система показала специальные предложения.
Альтернативные потоки: Нет.

Рис. 4.11. *Моделирование последовательности действий в потоке событий с помощью ключевого слова «Пока»*

Ключевым моментом является то, что альтернативные потоки часто не возвращаются в основной поток. Это происходит потому, что они обычно обрабатывают ошибки и исключения основного потока и имеют другие постусловия. Альтернативные потоки наглядно представлены на рис. 4.12.

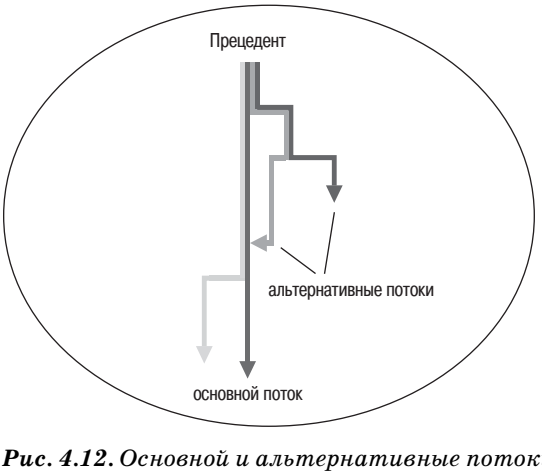


Рис. 4.12. *Основной и альтернативные потоки*

Прецедент: CreateNewCustomerAccount
ID: 5
Краткое описание: Система создает новую учетную запись для Покупателя.
Главные актеры: Покупатель
Второстепенные актеры: Нет.
Предусловия: Нет.
Основной поток: 1. Прецедент начинается, когда Покупатель выбирает опцию «создать новую учетную запись Покупателя». 2. Пока данные Покупателя недействительны. 2.1. Система просит Покупателя ввести его данные, включая адрес электронной почты, пароль и еще раз пароль для подтверждения. 2.2. Система проверяет действительность данных Покупателя. 3. Система создает новую учетную запись для Покупателя.
Постусловия: 1. Новая учетная запись создана для Покупателя.
Альтернативные потоки: InvalidEmailAddress InvalidPassword Cancel

Рис. 4.13. *Спецификация прецедента с альтернативными потоками*

Альтернативные потоки могут быть задокументированы отдельно или добавляться в конце прецедента. Мы предпочитаем документировать их отдельно.

В качестве примера модели прецедента с альтернативными потоками рассмотрим рис. 4.13.

Как видим, у этого прецедента три альтернативных потока: InvalidEmailAddress (недействительный адрес электронной почты), InvalidPassword (недействительный пароль) и Cancel (отмена). На рис. 4.14 задокументирован альтернативный поток InvalidEmailAddress.

Обратите внимание, что для ввода альтернативных потоков в шаблон прецедента было внесено несколько изменений:

- Имя – для альтернативных потоков используется следующая схема присваивания имен:

  Альтернативный поток: CreateNewCustomerAccount: InvalidEmailAddress

  Такое имя говорит о том, что это альтернативный поток InvalidEmailAddress для прецедента CreateNewCustomerAccount.

- ID – обратите внимание на применение иерархической системы нумерации для обеспечения связи альтернативного потока с основным прецедентом.

Альтернативный поток: CreateNewCustomerAccount:InvalidEmailAddress
ID: 5.1
Краткое описание: Система сообщает Покупателю, что он ввел недействительный адрес электронной почты.
Главные актеры: Покупатель
Второстепенные актеры: Нет.
Предусловия: 1. Покупатель ввел недействительный адрес электронной почты.
Альтернативные потоки: 1. Альтернативный поток начинается после шага 2.2 основного потока. 2. Система сообщает Покупателю, что он ввел недействительный адрес электронной почты.
Постусловия: Нет.

Рис. 4.14. *Альтернативный поток InvalidEmailAddress*

- Актеры – перечислены актеры, принимающие участие в альтернативном потоке.
- Предусловия и постусловия – альтернативные потоки могут иметь собственный набор предусловий и постусловий, отличный от набора прецедента. Если альтернативный поток возвращается в основной поток, его постусловия добавляются к постусловиям основного потока.
- Альтернативный поток – шаги альтернативного потока.
- У альтернативного потока *не* должно быть альтернативных потоков, иначе описание прецедента становится слишком запутанным.

Альтернативные потоки могут быть инициированы тремя разными способами:

1. Альтернативный поток может быть инициирован *вместо* основного потока.
2. Альтернативный поток может быть инициирован *после определенного этапа* основного потока.
3. Альтернативный поток может быть инициирован в *любой момент* в ходе выполнения основного потока.

Если альтернативный поток выполняется вместо основного потока, он инициируется главным актером и полностью замещает весь прецедент.

Если альтернативный поток инициируется после определенного этапа основного потока, он должен начинаться следующим образом:

1. Альтернативный поток начинается после шага X основного потока.

Альтернативный поток: CreateNewCustomerAccount:Cancel
ID: 5.2
Краткое описание: Покупатель отменяет процесс создания учетной записи.
Главные актеры: Покупатель
Второстепенные актеры: Нет.
Предусловия: Нет.
Альтернативные потоки: 1. Альтернативный поток начинается в любой момент времени. 2. Покупатель отменяет создание учетной записи.
Постусловия: 1. Новая учетная запись не была создана для Покупателя.

Рис. 4.15. *Альтернативный поток Cancel*

Такой поток – это форма ветвления. Она отличается от рассматриваемого в разделе 4.5.6.1 тем, что является значительным отклонением от основного потока и может в него больше не вернуться.

Если альтернативный поток может быть инициирован в любой момент во время выполнения основного потока, начинать его надо следующим образом:

1. Альтернативный поток начинается в любой момент времени.

Такие альтернативные потоки используются для моделирования того, что может произойти в любой точке основного потока до заключительного этапа. Например, в прецеденте CreateNewCustomerAccount Customer может отменить создание учетной записи в любой момент. Поток Cancel можно задокументировать, как показано на рис. 4.15.

Если альтернативный поток должен вернуться в основной, можно воспользоваться следующей формой записи:

N. Альтернативный поток возвращается на шаг M основного потока.

В этом примере альтернативный поток выполняет свой последний этап N и продолжается выполнение основного потока с этапа M.

4.5.7.1. Выявление альтернативных потоков

Чтобы выявить альтернативные потоки, нужно внимательно изучить основной поток. На каждом шаге основного потока необходимо искать:

- возможные альтернативы основному потоку;
- ошибки, которые могут возникнуть в основном потоке;
- прерывания, которые могут случиться в конкретной точке основного потока;

- прерывания, которые могут произойти в *любой* точке основного потока.

Каждый из перечисленных факторов является возможным источником альтернативного потока.

4.5.7.2. Сколько альтернативных потоков?

> Документируйте только самые важные альтернативные потоки.

Как было сказано, в прецеденте всегда есть один основной поток. Однако наряду с основным может быть *много* альтернативных потоков. Вопрос: «Сколько?» Надо свести число альтернативных потоков до необходимого минимума. Есть две стратегии.

- Выбрать самые важные альтернативные потоки и задокументировать только их.
- Если существуют группы очень сходных альтернативных потоков, документировать один из них как образец и (если необходимо) добавить примечания, объясняющие, чем остальные потоки отличаются от образца.

Вернемся к аналогии с дельтой реки. Помимо основного рукава в дельте может образоваться много ветвящихся и извилистых альтернативных рукавов. Нанести на карту все рукава невозможно, поэтому выбираются только наиболее заметные. Также многие из этих ответвлений направлены практически в одну сторону и имеют лишь небольшие отличия. Поэтому можно подробно изобразить на карте только один рукав и составить примечания, объясняющие, чем отличаются остальные меньшие рукава. В этом состоит рациональный и эффективный способ моделирования сложного прецедента.

Основной принцип в моделировании прецедентов – обеспечить *необходимый минимум* информации. Это значит, что многие альтернативные потоки могут вообще никогда не входить в спецификацию. Для понимания функционирования системы может оказаться вполне достаточным краткое описание альтернативных потоков в виде одной строчки. Это важный момент. Очень легко увязнуть в альтернативных потоках. Не один процесс моделирования прецедентов провалился из-за этого.

Необходимо помнить, что прецеденты выявляются, чтобы *понять требуемое поведение* системы, а не с целью создания полной модели прецедентов. Поэтому, когда достигнуто понимание, моделирование прецедентов должно быть остановлено. Кроме того, поскольку UP является итеративным жизненным циклом, всегда можно вернуться к прецедентам и доработать их, если возникли некоторые не до конца понятные аспекты поведения системы.

4.6. Отображение требований

> При отображении требований устанавливаются взаимосвязи между моделью требований и моделью прецедентов.

Модель требований и модель прецедентов фактически обеспечивают две «базы данных» функциональных требований. Важно сопоставить эти две модели, чтобы выяснить, нет ли в одной из них чего-то, что не охвачено в другой, и наоборот. Такая постановка вопроса – один из аспектов отображения требований.

Отображение функциональных требований осложняется тем фактом, что между отдельными функциональными требованиями и прецедентами установлены отношения «многие-ко-многим». Один прецедент будет охватывать множество отдельных функциональных требований, и одно функциональное требование может появляться в нескольких разных прецедентах.

Надеемся, в вашем распоряжении будут инструменты для моделирования, имеющие поддержку отображения требований. Такие инструментальные средства для выработки требований, как RequisitePro и DOORS позволяют связывать отдельные требования в базе данных требований с конкретными прецедентами, и наоборот. Кстати, UML предоставляет достаточно хорошую поддержку отображения требований. С помощью помеченных значений можно ассоциировать список ID требований с каждым прецедентом. В инструменте выработки требований можно связать один или более идентификаторов прецедентов с конкретными требованиями.

В случае отсутствия такой поддержки в инструменте для моделирования всю эту работу необходимо выполнять вручную. Для этого полезно создать матрицу отображаемости требований. Это простая таблица с номерами ID отдельных требований, расположенными по вертикали, и именами прецедентов (и/или номерами ID) – по горизонтали. Во всех ячейках, где прецедент и требование пересекаются, ставится крестик. Обычно матрицы прослеживания требований создаются в виде электронных таблиц. Пример приведен в табл. 4.2.

Таблица 4.2

		Прецедент			
		$П_1$	**$П_2$**	**$П_3$**	**$П_4$**
Требование	T1	X			
	T2		X	X	
	T3			X	
	T4				X
	T5	X			

Матрица отображаемости требований – полезный инструмент для проверки согласованности. Если существует требование, не отображающееся ни в один прецедент, значит, упущен прецедент. И наоборот, если есть прецедент, которому не поставлено в соответствие ни одно требование, понятно, что набор требований неполный.

С помощью комплекта инструментов SUMR, который обсуждался в разделе 2.2, можно автоматизировать создание матрицы отображаемости потенциальных требований. Идея проста: если термин глоссария проекта встречается и в требовании, и в прецеденте, велика вероятность того, что они как-то связаны между собой. Так создается матрица прослеживания предполагаемых требований. Мы говорим «предполагаемых», потому что в результате такого простого текстового анализа могут появиться ошибки и упущения. Эта матрица нуждается в ручной доработке. Тем не менее она может существенно сэкономить время и помочь разработчикам требований решить трудные задачи, которые в противном случае могли бы быть вообще не реализованы.

4.7. Когда применять моделирование прецедентов

> Прецеденты хорошо применять для определения функциональности системы. Они плохо подходят для выявления ограничений системы.

Прецеденты фиксируют функциональные требования и поэтому не эффективны для систем, в которых доминируют нефункциональные требования.

Прецеденты являются лучшим выбором для фиксирования требований в тех случаях, когда:

- в системе преобладают функциональные требования;
- в системе много типов пользователей, которым она предоставляет разные функциональные возможности (много актеров);
- в системе много интерфейсов (много актеров).

Прецеденты не стоит применять в тех случаях, когда:

- в системе преобладают нефункциональные требования;
- в системе мало пользователей;
- в системе мало интерфейсов.

Примерами систем, для которых не годятся прецеденты, являются встроенные (embedded) системы и системы со сложным алгоритмом, но с малым количеством интерфейсов. Для моделирования таких систем намного лучше воспользоваться более традиционными методами выработки требований. Главное – правильно выбрать инструментальное средство.

4.8. Что мы узнали

Эта глава была посвящена определению требований, предъявляемых к системе, путем моделирования прецедентов. Мы узнали следующее.

- Деятельность по моделированию прецедентов является частью рабочего потока определения требований.
- Большая часть работы в рабочем потоке определения требований осуществляется в фазах Начало и Уточнение жизненного цикла UP проекта.
- Основные деятельности UP – Выявление актеров и прецедентов и Детализация прецедента.
- Моделирование прецедентов – еще одна форма выработки требований, которая происходит следующим образом:
  - выявление контекста;
  - выявление актеров;
  - выявление прецедентов.
- Контекст системы определяет, что является частью системы, а что находится вне системы.
- Актеры – это роли, выполняемые сущностями, внешними по отношению к системе, которые взаимодействуют непосредственно с системой.
  - Выявить актеров можно, выяснив, кто или что использует или взаимодействует непосредственно с системой.
  - Часто время является актером.
- Прецеденты – это функции, осуществляемые системой с точки зрения конкретных актеров; их цель – принести пользу этим актерам. Для выявления прецедентов необходимо выяснить, как каждый актер взаимодействует с системой.
  - Прецеденты можно выявить, рассмотрев, какие функциональные возможности система предлагает актерам.
  - Прецеденты всегда инициируются актером.
  - Прецеденты всегда пишутся с точки зрения актеров.
- На диаграмме прецедентов отражены:
  - контекст;
  - актеры;
  - прецеденты;
  - взаимодействия.
- Глоссарий проекта предоставляет определения ключевых бизнес-терминов, включает синонимы и омонимы.

- Спецификация прецедента включает:
  - имя прецедента;
  - уникальный идентификатор;
  - краткое описание – цель прецедента;
  - актеров:
    - главных актеров – инициируют прецедент;
    - второстепенных актеров – взаимодействуют с прецедентом после его инициации.
  - предусловия – ограничения системы, которые оказывают влияние на выполнение прецедента;
  - основной поток – последовательность декларативных, упорядоченных во времени шагов прецедента;
  - постусловия – ограничения системы, возникающие в результате выполнения прецедента;
  - альтернативные потоки – список альтернатив основному потоку.
- Можно сократить число прецедентов, позволив ограниченное количество ветвлений в рамках основного потока событий. Для этого:
  - применяйте ключевое слово Если (If) для ветвлений, возникающих на конкретных шагах потока;
  - ветвления, которые могут возникнуть на любом шаге основного потока, помещайте в секцию Альтернативный поток в описании прецедента.
- Повторения в рамках потока можно показать с помощью ключевых слов:
  - Для (выражение, описывающее итерации);
  - Пока (логическое условие).
- У каждого прецедента есть один основной поток – «идеальный» сценарий, когда все идет так, как запланировано.
- Более сложные прецеденты могут иметь один или более альтернативных потоков. Это пути в прецеденте, представляющие исключения, ответвления и прерывания.
- Ключевые альтернативные потоки выявляются путем анализа основного потока и поиска:
  - альтернатив;
  - ситуаций, связанных с появлением ошибки;
  - прерываний.
- Прецедент необходимо расщеплять на альтернативные потоки, только если это повышает ценность модели.
- Требования в модели требований могут быть сопоставлены с прецедентами при помощи матрицы отображаемости требований.

- Моделирование прецедентов лучше всего подходит для систем, в которых:
  - преобладают функциональные требования;
  - много типов пользователей;
  - много интерфейсов для взаимодействия с другими системами.
- Моделирование прецедентов меньше подходит для систем, в которых:
  - преобладают нефункциональные требования;
  - мало пользователей;
  - мало интерфейсов.

5

Дополнительные аспекты моделирования прецедентов

5.1. План главы

В настоящей главе рассматриваются некоторые дополнительные аспекты моделирования прецедентов. И в завершение приводятся несколько советов и рекомендаций.

Мы обсудим отношения, которые могут возникать между актерами и актерами и между прецедентами и прецедентами. К ним относятся:

- обобщение актеров – отношение обобщения между обобщенным актером и конкретным актером;
- обобщение прецедентов – отношение обобщения между обобщенным прецедентом и специализированным прецедентом;
- «include» (включить) – отношение между прецедентами, которое позволяет одному прецеденту включать в себя поведение другого;
- «extend» (расширить) – отношение между прецедентами, которое позволяет одному прецеденту расширять свое поведение одним или более фрагментами поведения другого.

Важно сохранять максимальную простоту модели, поэтому эти отношения надо применять аккуратно и только там, где они действительно делают модель прецедентов более понятной. Можно легко увлечься «include» и «extend», но необходимо избегать этого.

5.2. Обобщение актеров

В примере на рис. 5.2 между двумя актерами, Customer (клиент) и SalesAgent (торговый агент), можно найти много общего в том, как они взаимодействуют с системой Sales (товарооборот) (здесь SalesAgent может управлять куплей-продажей от имени Customer). Оба актера инициируют прецеденты ListProducts (представить список продуктов), OrderProducts

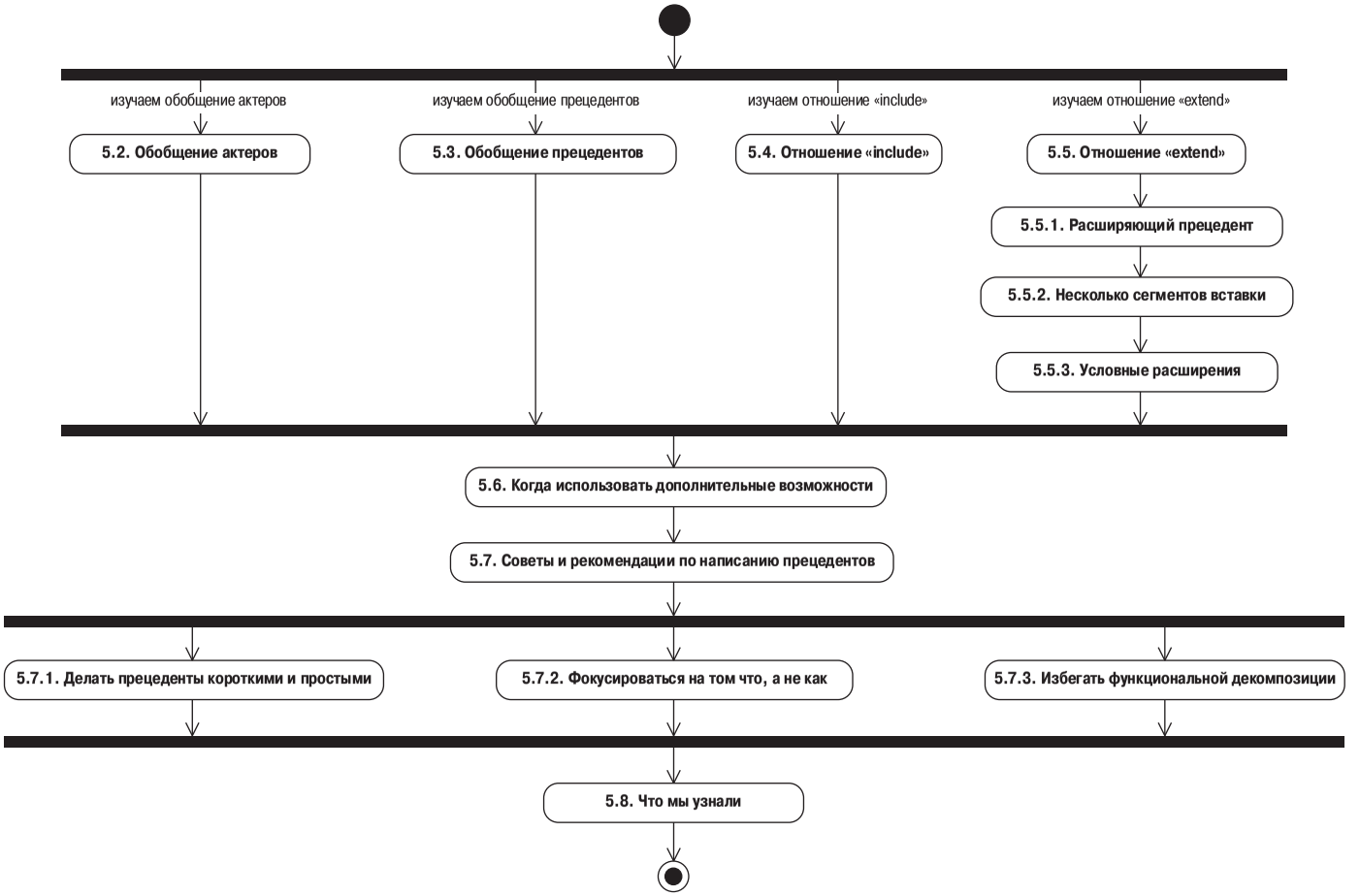


Рис. 5.1. *План главы*

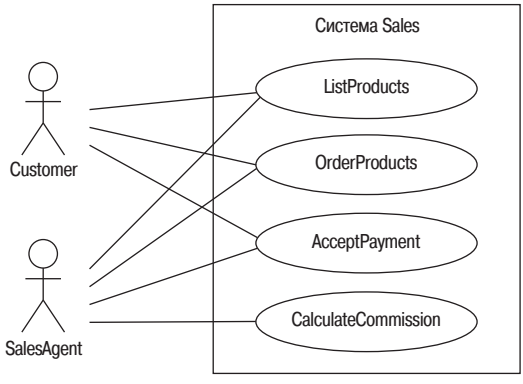


Рис. 5.2. *Пример двух актеров с общим поведением*

(заказать продукты) и AcceptPayment (принять платеж). По сути, единственное отличие между ними в том, что SalesAgent может инициировать прецедент CalculateCommission (вычислить комиссию). Кроме того, из-за сходства поведения этих актеров на диаграмме возникает масса пересекающихся линий. Это указывает на наличие некоего общего поведения, которое может быть вынесено и представлено в виде более *обобщенного* актера.

> Обобщение актеров выносит поведение, общее для двух или более актеров, в актера-родителя.

Общее поведение можно вынести путем обобщения актеров (рис. 5.3). Тем самым мы создаем абстрактного актера, называемого Purchaser (покупатель), который взаимодействует с прецедентами ListProducts, OrderProducts и AcceptPayment. Customer и SalesAgent – это конкретные актеры, потому что данные роли могут выполнять реальные люди (или другие системы). Purchaser – абстрактный актер, поскольку он является абстракцией, введенной просто для представления общего поведения (возможности делать покупки) двух конкретных актеров.

Customer и SalesAgent наследуют все роли и отношения с прецедентами своего абстрактного родителя. Таким образом, как видно из рис. 5.3, и Customer, и SalesAgent взаимодействуют с прецедентами ListProducts, OrderProducts и AcceptPayment. Они наследуют это от своего родителя, Purchaser. Кроме того, SalesAgent взаимодействует с прецедентом CalculateCommission. Это поведение не является унаследованным, оно характерно для актера SalesAgent. Как видим, разумное использование абстрактных актеров может упростить диаграмму прецедентов. Это также упрощает семантику модели прецедентов, потому что появляется возможность интерпретировать разные вещи одинаково.

Следует отметить, что актер-родитель в обобщении актеров не всегда абстрактен. Это может быть конкретная роль, исполняемая человеком

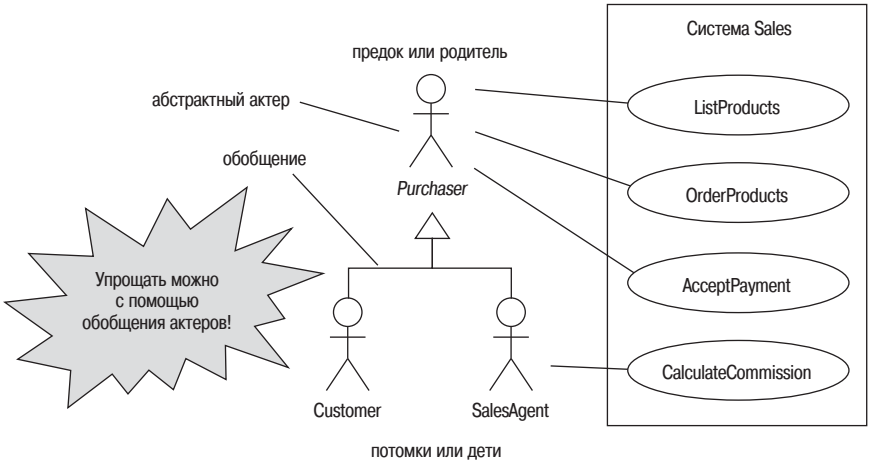


Рис. 5.3. *Общее поведение вынесено в актера-родителя*

или системой. Однако хорошим стилем считается делать актера-родителя абстрактным для сохранения простоты семантики обобщения.

> Актер-потомок может использоваться везде, где ожидается актер-предок.

Мы увидели, что если два актера одинаково общаются с одним и тем же набором прецедентов, их можно обобщить в другом (возможно, абстрактном) актере. Актеры-потомки наследуют роли и отношения с прецедентами от актера-родителя. Актер-потомок может использоваться вместо актера-предка. Это принцип замещаемости, с помощью которого можно проверить правильность использования обобщения для *любого* классификатора.

В данном примере SalesAgent или Customer могут использоваться вместо Purchaser везде (т. е. взаимодействовать с прецедентами ListProducts, OrderProducts и AcceptPayment). Таким образом, обобщение актеров является правильной стратегией.

5.3. Обобщение прецедентов

Обобщение прецедентов используется, если есть один или более прецедентов, которые на самом деле являются специализациями более общего прецедента. Как и обобщение актеров, этот прием следует применять, только если он упрощает модель прецедентов.

> Обобщение прецедентов выносит поведение, общее для одного или более прецедентов, в родительский прецедент.

В обобщении прецедентов дочерние прецеденты представляют более специализированные формы их родителей. Потомки могут:

- наследовать возможности родительского прецедента;
- вводить новые возможности;
- переопределять (менять) унаследованные возможности.

Дочерний прецедент автоматически наследует *все* возможности своего родителя. Однако не все возможности прецедента могут быть переопределены. Ограничения приведены в табл. 5.1.

Таблица 5.1

Возможность прецедента	Наследование	Добавление	Переопределение
Отношение	Да	Да	Нет
Точка расширения	Да	Да	Нет
Предусловие	Да	Да	Да
Постусловие	Да	Да	Да
Шаг основного потока	Да	Да	Да
Альтернативный поток	Да	Да	Да

В UML 1.5 прецеденты имели атрибуты и операции, в UML 2 их нет. Фактически атрибуты и операции прецедентов не имели особого значения, они редко использовались и редко поддерживались инструментальными средствами UML. Согласно спецификации UML 1.5, операции прецедента даже не могли быть запрошены извне, поэтому трудно представить, зачем они вообще были нужны.

Итак, как же осуществляется документирование обобщения прецедентов в описаниях прецедентов? Спецификация UML по этому поводу безмолвствует. Однако существует несколько довольно стандартных методов. Мы предпочитаем использовать простой язык тегов для обозначения пяти возможностей в дочернем прецеденте. Есть два правила применения этого метода:

- Каждый номер шага в потомке сопровождается номером эквивалентного шага родителя, если таковой имеется.
  Например: 1. (2.). Некоторый шаг.
- Если шаг потомка переопределяет шаг родителя, его номер сопровождается буквой «о» (что значит overridden – переопределенный) и родительским номером шага. Например: 6. (об.) Другой шаг.

В табл. 5.2 представлен синтаксис всех пяти возможных вариантов.

На рис. 5.4 показан фрагмент диаграммы прецедентов системы Sales. Здесь есть родительский прецедент FindProduct (найти продукт) и две его специализации: FindBook (найти книгу) и FindCD (найти CD).

На рис. 5.5 показано описание родительского прецедента FindProduct. Обратите внимание на ее очень высокий уровень абстракции.

Таблица 5.2

Возможность...	Пример обозначения
Унаследована без изменений	3. (3.) Покупатель вводит запрашиваемый критерий.
Унаследована и перенумерована	6.2. (6.1.) Система сообщает Покупателю, что соответствующие продукты не найдены.
Унаследована и переопределена	1. (o1.) Покупатель выбирает опцию «найти книгу».
Унаследована, переопределена и перенумерована	5.2. (o5.1.) Система выводит на экран страницу с данными максимум пяти книг.
Добавлена	6.3. Система повторно выводит на экран страницу поиска «найти книгу».

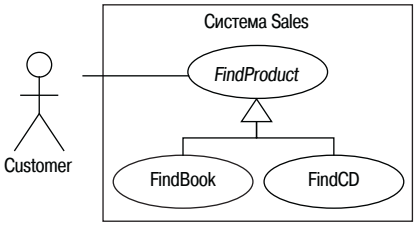


Рис. 5.4. *Фрагмент диаграммы прецедентов системы Sales*

Прецедент: *FindProduct*
ID: 6
Краткое описание: Customer ищет продукт.
Главные актеры: Customer
Второстепенные актеры: Нет.
Предусловия: Нет.
Основной поток: 1. Customer выбирает опцию «найти продукт». 2. Система запрашивает у Customer критерий поиска. 3. Customer вводит запрашиваемый критерий. 4. Система ищет продукты, соответствующие критерию от Customer. 5. Если система находит соответствующие продукты 5.1. Система выводит на экран список соответствующих продуктов. 6. Иначе 6.1. Система сообщает Customer о том, что соответствующие продукты не найдены.
Постусловия: Нет.
Альтернативные потоки: Нет.

Рис. 5.5. *Описание родительского прецедента FindProduct*

	Прецедент: FindBook
	ID: 7
	ID родителя: 6
	Краткое описание: Customer ищет книгу.
	Главные актеры: Customer
	Второстепенные актеры: Нет.
	Предусловия: Нет.
	Основной поток:
переопределенный	1. (o1.) Customer выбирает опцию «найти книгу».
переопределенный	2. (o2.) Система запрашивает у Customer критерий поиска книги, включающий автора, название, ISBN или тематику.
унаследованный без изменений	3. (3.) Customer вводит запрашиваемый критерий.
переопределенный	4. (o4.) Система ищет книги, соответствующие критерию от Customer.
переопределенный	5. (o5.) Если система находит соответствующие книги.
добавленный	5.1. Система выводит на экран текущий бестселлер.
переопределенный и перенумерованный	5.2. (o5.1.) Система выводит на экран данные по максимум пяти книгам.
добавленный	5.3. Для каждой книги система выводит название, автора, цену и ISBN.
добавленный	5.4. Пока есть другие соответствующие запросу книги, система предоставляет Customer опцию для отображения следующей страницы с книгами.
унаследованный без изменений	6. (6.) Иначе
добавленный	6.1. Система выводит на экран текущий бестселлер.
перенумерованный	6.2. (6.1.) Система сообщает Покупателю о том, что соответствующие книги не найдены.
	Постусловия: Нет.
	Альтернативные потоки: Нет.

Рис. 5.6. *Описание дочернего прецедента FindBook*

Один из дочерних прецедентов, FindBook, показан на рис. 5.6. Здесь демонстрируется применение нашего стандарта обозначения переопределенных или новых возможностей. Как видно из рис. 5.6, дочерний прецедент FindBook намного более конкретный. В нем более абстрактный родитель специализирован для работы с конкретным типом продуктов – книгами.

Если в родительском прецеденте нет потока событий или поток событий не завершен, это абстрактный прецедент. Абстрактные прецеденты довольно широко распространены, потому что могут использоваться для описания поведения на самых высоких уровнях абстракции. Поскольку в абстрактных прецедентах поток событий отсутствует или является неполным, они не могут быть выполнены системой. Вместо потока событий в абстрактных прецедентах используется простое высокоуровневое текстовое описание поведения, которое должны реализовать их потомки. Это описание можно поместить в раздел «Краткое описание».

Как мы уже видели, унаследованные возможности в дочерних прецедентах показать сложно. Приходится применять определенный язык тегов или соглашение об обозначениях, что обычно сбивает с толку за-

интересованные стороны. Поскольку прецеденты предназначены для общения с заказчиками, это является серьезной проблемой. Другой недостаток состоит в том, что приходится вручную изменять таблицу соответствия между родителями и потомками, если один из них меняется. Это утомительно и может приводить к большому числу ошибок.

Одно из решений данной проблемы – ограничить родительский прецедент так, чтобы в нем не было основного потока, а только краткое описание семантики. Тогда не надо беспокоиться о наследовании или переопределении. В этом случае с помощью прецедентов можно легко и эффективно показать, что один или более прецедентов на самом деле являются просто особыми вариантами более общего прецедента. Более общий прецедент позволяет рассматривать систему более абстрактно и может указать на возможности оптимизации системы ПО.

5.4. Отношение «include»

Иногда в прецедентах присутствует многократное описание одних и тех же действий. Например, рассмотрим систему Personnel (персонал) (рис. 5.7). Практически любое действие системы начинается с получения данных о конкретном служащем. Если бы эту последовательность событий приходилось писать каждый раз, когда необходимы данные служащего, прецеденты имели бы повторяющиеся части. Отношение «include», устанавливаемое между прецедентами, позволяет включить поведение одного прецедента в поток другого прецедента.

> Отношение «include» выносит шаги, общие для нескольких прецедентов, в отдельный прецедент, который потом включается в остальные.

Включающий прецедент мы называем базовым, а тот прецедент, который включается, *включаемым*. *Включаемый* прецедент предоставляет поведение своему базовому прецеденту.

В базовом прецеденте необходимо *точно* указать место, где должно быть включено поведение включаемого прецедента. Синтаксис и семантика отношения «include» немного напоминают вызов функции.

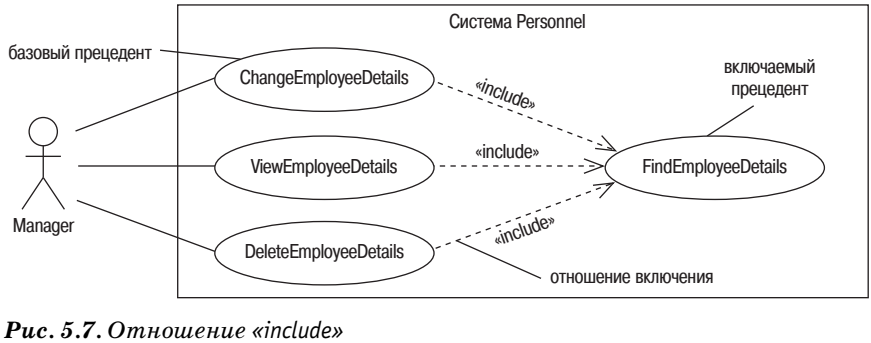


Рис. 5.7. *Отношение «include»*

Прецедент: ChangeEmployeeDetails	Прецедент: ViewEmployeeDetails	Прецедент: DeleteEmployeeDetails
ID: 1	ID: 2	ID: 3
Краткое описание: Manager меняет данные служащего.	Краткое описание: Manager просматривает данные служащего.	Краткое описание: Manager удаляет данные служащего.
Главные актеры: Manager	Главные актеры: Manager	Главные актеры: Manager
Второстепенные актеры: Нет.	Второстепенные актеры: Нет.	Второстепенные актеры: Нет.
Предусловия: 1. Manager входит в систему.	Предусловия: 1. Manager входит в систему.	Предусловия: 1. Manager входит в систему.
Основной поток: 1. include (FindEmployeeDetails). 2. Система выводит данные служащего. 3. Manager меняет данные служащего. ...	Основной поток: 1. include (FindEmployeeDetails). 2. Система выводит данные служащего. ...	Основной поток: 1. include (FindEmployeeDetails). 2. Система выводит данные служащего. 3. Manager удаляет данные служащего. ...
Постусловия: 1. Данные служащего изменены.	Постусловия: 1. Система вывела на экран данные служащего.	Постусловия: 1. Данные служащего удалены.
Альтернативные потоки: Нет.	Альтернативные потоки: Нет.	Альтернативные потоки: Нет.

Рис. 5.8. *Семантика отношения «include»*

Отношение «include» имеет простую семантику (рис. 5.8). Базовый прецедент выполняется до момента включения. Затем выполнение переходит во включаемый прецедент. По завершении включаемого прецедента управление вновь возвращается в базовый прецедент.

Базовый прецедент является незавершенным без всех его включаемых прецедентов. Они – неотъемлемые части базового прецедента. Однако включаемые прецеденты могут быть как полными, так и неполными. Если включаемый прецедент *неполный*, он просто содержит часть потока событий, которая имеет смысл только тогда, когда включена в соответствующий базовый прецедент. Обычно такие прецеденты называют фрагментом поведения. В этом случае говорят, что экземпляр включаемого прецедента не может быть создан, т. е. он не может быть инициирован актерами напрямую. Он может выполняться, только если он включен в соответствующий базовый прецедент. Однако если включаемый прецедент *полный*, он ведет себя как обычный прецедент. Его экземпляры могут быть созданы, и он может инициироваться актерами. Включаемый прецедент приведен на рис. 5.9. Он является неполным и, следовательно, его экземпляры не могут быть созданы.

5.5. Отношение «extend»

Отношение «extend» – способ введения нового поведения в существующий прецедент.

Прецедент: FindEmployeeDetails
ID: 4
Краткое описание: Manager ищет данные служащего.
Главные актеры: Manager
Второстепенные актеры: Нет.
Предусловия: 1. Manager входит в систему.
Основной поток: 1. Manager вводит ID служащего. 2. Система ищет данные служащего.
Постусловия: 1. Система нашла данные служащего.
Альтернативные потоки: Нет.

Рис. 5.9. *Пример неполного включаемого прецедента*

Отношение «extend» предоставляет возможность ввести новое поведение в существующий прецедент (рис. 5.10). Базовый прецедент предоставляет набор точек расширения (extension points) – точек входа, в которые может быть добавлено новое поведение. А расширяющий прецедент предоставляет ряд сегментов вставки, которые можно ввести в базовый прецедент в места, указанные точками входа. Как вскоре будет показано, отношение «extend» может использоваться для того, чтобы *точно* указать, какие *именно* точки расширения базового прецедента подлежат расширению.

В отношении «extend» любопытно то, что базовый прецедент ничего не знает о расширяющих прецедентах, он просто предоставляет для них точки входа. Базовый прецедент абсолютно полон и без расширений. Это *существенно* отличает «extend» от отношения «include», где базовые

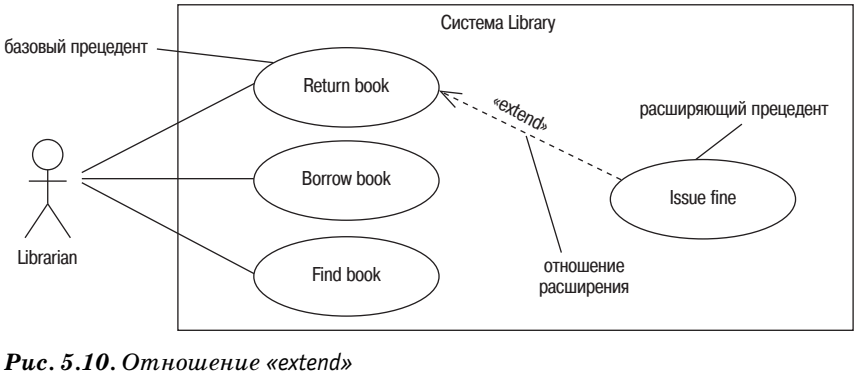


Рис. 5.10. *Отношение «extend»*

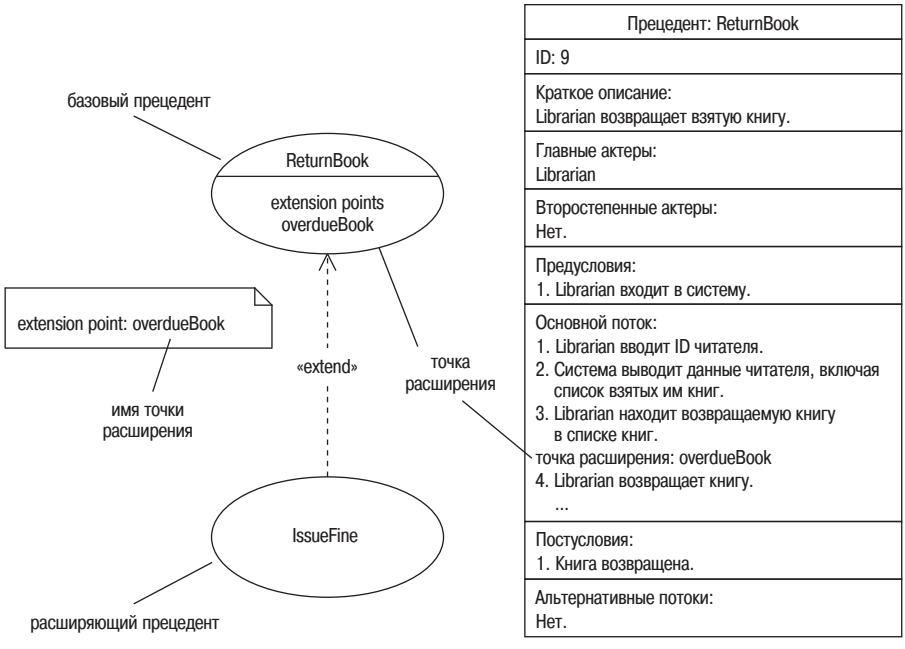


Рис. 5.11. *Обозначение точек расширения*

прецеденты остаются неполными без включаемых прецедентов. Более того, точки расширения на самом деле не вводятся в поток событий базового прецедента; они накладываются поверх потока.

Точки расширения обозначаются в потоке событий базового прецедента, как показано на рис. 5.11. Точки расширения также можно показать на диаграмме прецедентов, перечислив их в новой ячейке пиктограммы базового прецедента.

Обратите внимание на то, что точки расширения в основном потоке не пронумерованы. Они появляются *между* пронумерованными шагами потока. UML явно определяет тот факт, что точки расширения фактически существуют в слое поверх основного потока. Следовательно, они вообще не являются его частью. Этот слой подобен прозрачной пленке, наложенной поверх основного потока, на которую нанесены точки расширения. Основная идея введения этого слоя – сделать поток базового прецедента полностью независимым от точек расширения. Иначе говоря, поток базового прецедента не знает (или не интересуется), в каких точках происходит его расширение. Это позволяет использовать отношение «extend» для создания произвольных или специальных расширений потока базового прецедента.

При использовании «extend» базовый прецедент выступает в роли модульного каркаса, к которому можно подключать расширения в предопределенных точках расширения. В примере на рис. 5.11 в базовом

прецеденте ReturnBook точка расширения overdueBook находится между шагами 3 и 4 потока событий.

Отношение «extend» предоставляет хороший способ обработки исключительных ситуаций или ситуаций, когда нужен гибкий каркас, поскольку невозможно предсказать (или просто не известны) все возможные расширения.

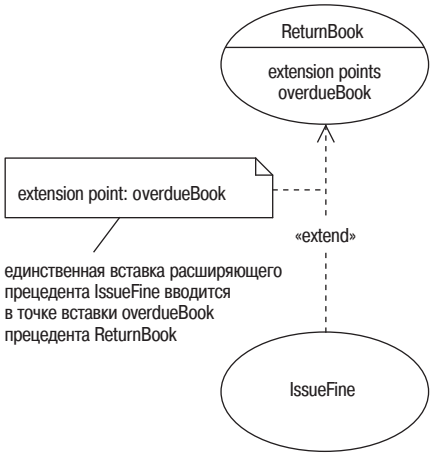
5.5.1. Расширяющий прецедент

Расширяющие прецеденты *обычно* не являются полными прецедентами, поэтому, как правило, их экземпляр не может быть создан. Обычно они состоят всего из одного или нескольких фрагментов поведения, называемых сегментами вставки. Отношение «extend» определяет точку расширения в базовом прецеденте, в которой будет введен сегмент вставки. Здесь действуют следующие правила:

- Отношение «extend» должно определять одну или несколько точек расширения базового прецедента. В противном случае предполагается, что отношение «extend» относится ко *всем* точкам расширения.
- В расширяющем прецеденте должно быть столько же сегментов вставки, сколько точек расширения определено в отношении «extend».
- Два расширяющих прецедента могут «расширять» один базовый прецедент в одной и той же точке расширения. Но в этом случае порядок выполнения расширений не определен.

В примере на рис. 5.12 в расширяющем прецеденте всего один сегмент вставки IssueFine (назначить штраф).

Расширяющий прецедент может также иметь предусловия и постусловия. Предусловия должны быть выполнены, в противном случае сег-



Расширяющий прецедент: IssueFine
ID: 10
Краткое описание: Сегмент 1: Librarian записывает и распечатывает штраф.
Главные актеры: Librarian
Второстепенные актеры: Нет.
Сегмент 1 предусловия: 1. Возвращенная книга просрочена.
Поток сегмента 1: 1. Librarian вводит данные штрафа в систему. 2. Система распечатывает штраф.
Сегмент 1 постусловия: 1. Штраф записан в системе. 2. Система распечатала штраф.

Рис. 5.12. *Расширяющий прецедент с одним сегментом вставки*

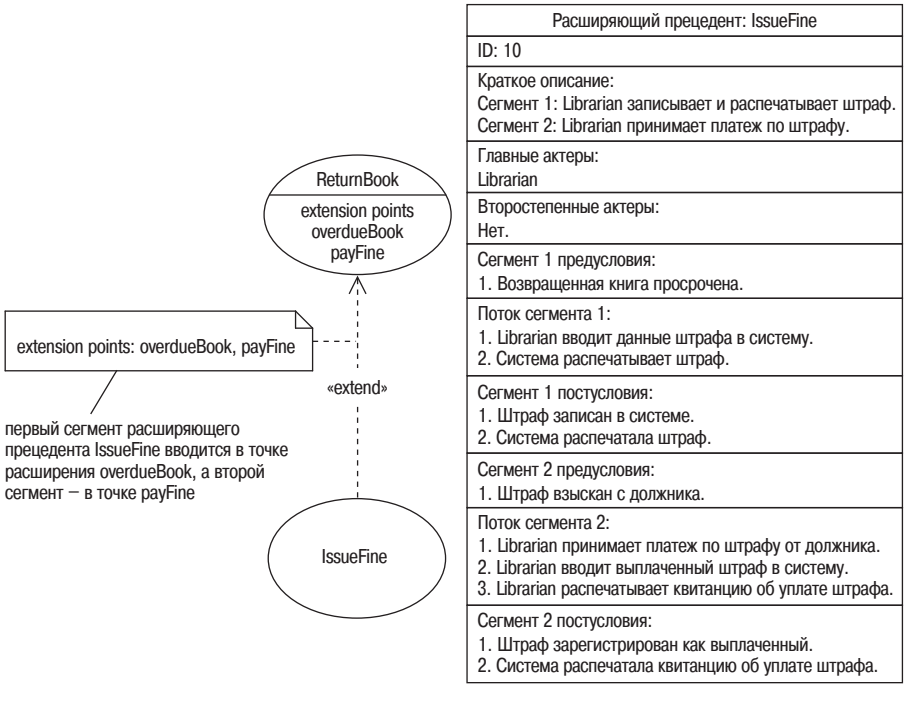
мент не выполняется. Постусловия ограничивают состояние системы после выполнения сегмента.

У самих расширяющих прецедентов могут быть расширяющие или включаемые прецеденты. Однако лучше избегать такой вложенности, поскольку это сильно усложняет систему.

5.5.2. Несколько сегментов вставки

В расширяющем прецеденте может быть несколько сегментов вставки. Это полезно в тех случаях, когда не получается полностью реализовать расширение в одном сегменте из-за того, что необходимо вернуться и что-то сделать в основном потоке базового прецедента. Обратимся к примеру, изображенному на рис. 5.13. Пусть после записи и распечатки штрафа выполнение возвращается в основной поток для обработки других просроченных книг, после чего в точке расширения payFine (выплатить штраф) должнику предлагается заплатить общую сумму штрафа. Конечно, такая процедура более эффективна, чем взимание платежей по каждому штрафу в отдельности (так произошло бы, если бы эти два сегмента были сведены в один сегмент IssueFine).

При создании расширяющих прецедентов с несколькими сегментами необходимо четко нумеровать каждый сегмент, как показано на рис. 5.13, поскольку здесь важен порядок сегментов: первый сегмент



Расширяющий прецедент: IssueFine
ID: 10
Краткое описание: Сегмент 1: Librarian записывает и распечатывает штраф. Сегмент 2: Librarian принимает платеж по штрафу.
Главные актеры: Librarian
Второстепенные актеры: Нет.
Сегмент 1 предусловия: 1. Возвращенная книга просрочена.
Поток сегмента 1: 1. Librarian вводит данные штрафа в систему. 2. Система распечатывает штраф.
Сегмент 1 постусловия: 1. Штраф записан в системе. 2. Система распечатала штраф.
Сегмент 2 предусловия: 1. Штраф взыскан с должника.
Поток сегмента 2: 1. Librarian принимает платеж по штрафу от должника. 2. Librarian вводит выплаченный штраф в систему. 3. Librarian распечатывает квитанцию об уплате штрафа.
Сегмент 2 постусловия: 1. Штраф зарегистрирован как выплаченный. 2. Система распечатала квитанцию об уплате штрафа.

Рис. 5.13. *Расширяющий прецедент с двумя сегментами вставки*

вставляется в первой точке расширения и т. д. Поэтому надо очень аккуратно записывать сегменты в правильном порядке, а затем придерживаться его.

5.5.3. Условные расширения

Пример на рис. 5.14 демонстрирует другой, более гуманный подход к должнику: если он впервые задержал книгу, выдается предупреждение, а штраф выписывается только при повторном нарушении правил. Это можно смоделировать путем добавления нового расширяющего прецедента IssueWarning (выдать предупреждение) и введения условий в отношения «extend». Условия (conditions) – это логические выражения. Вставка осуществляется только в том случае, если выражение истинно.

Обратите внимание, что расширяющий прецедент IssueWarning вводится только в точке расширения overdueBook (просроченная книга). Однако (как и ранее) расширяющий прецедент IssueFine вводится и в точке

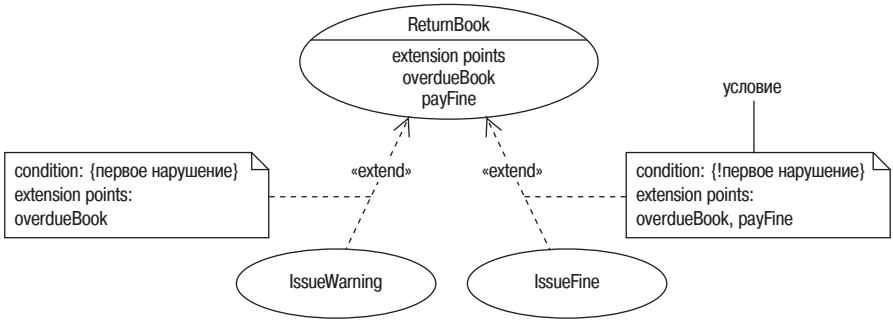


Рис. 5.14. *Добавление условного расширяющего прецедента*

Расширяющий прецедент: IssueWarning
ID: 11
Краткое описание: Сегмент 1: Librarian выдает предупреждение.
Главные актеры: Librarian
Второстепенные актеры: Нет.
Сегмент 1 предусловия: 1. Возвращенная книга просрочена.
Поток сегмента 1: 1. Librarian вводит данные предупреждения в систему.
Сегмент 1 постусловия: 1. Предупреждение записано в системе.

Рис. 5.15. *Описание условного расширяющего прецедента*

overdueBook, и в точке payFine (выплатить штраф). Это свидетельствует о том, что IssueWarning (рис. 5.15) содержит только один сегмент вставки, тогда как IssueFine (как мы уже видели) – два.

5.6. Когда применять дополнительные возможности

> Применяйте дополнительные возможности, только если они упрощают модель и делают ее более понятной.

Применяйте дополнительные возможности, только если они упрощают модель прецедентов. Мы вновь убеждаемся в том, что лучшие модели прецедентов – это простые модели. Запомните, что модель прецедентов – это изложение требований, то есть она должна быть понятной не только разработчикам моделей, но и заинтересованным сторонам. Простая модель прецедентов, в которой дополнительные возможности применяются редко или вообще отсутствуют, предпочтительнее модели, переполненной дополнительными возможностями, даже если последняя кажется разработчику более изысканной.

Учитывая опыт моделирования прецедентов в различных компаниях, можно сделать следующие выводы:

- обычно заинтересованные стороны после небольшой тренировки и обучения могут без труда разбираться в актерах и прецедентах;
- заинтересованным сторонам сложнее воспринимать обобщение актеров;
- широкое использование отношения «include» затрудняет понимание моделей прецедентов – заинтересованным сторонам и разработчикам моделей приходится рассматривать несколько прецедентов для получения полной картины;
- у заинтересованных сторон возникают большие сложности с пониманием отношения «extend» даже после подробных объяснений;
- как это ни удивительно, многие разработчики объектных моделей неверно понимают семантику отношения «extend»;
- обобщение прецедентов следует применять, только если в системе используются абстрактные (а не конкретные) родительские прецеденты, в противном случае это сильно усложняет дочерние прецеденты.

5.7. Советы и рекомендации по написанию прецедентов

В данном разделе предлагается несколько советов и рекомендаций по написанию прецедентов.

5.7.1. Делайте прецеденты короткими и простыми

Основной девиз: «Делайте прецеденты короткими, делайте их простыми». Они должны включать именно такой объем информации, которого достаточно для записи требований. К сожалению, в некоторых проектах разработчики избегают простого и сжатого изложения и стремятся к непомерному увеличению объема документации. Гради Буч называет эту тенденцию «бумажной жадностью».

Есть хорошее правило: основной поток прецедента должен помещаться на одной странице. Немного больше, и прецедент практически наверняка окажется слишком длинным. В реальности большинство прецедентов короче половины страницы.

Начать надо с упрощения фраз (использовать только короткие повествовательные предложения, как описано в разделе 3.6.2). Затем удалить все детали проектирования (см. следующий раздел). Если прецедент по-прежнему слишком велик, необходимо повторно проанализировать проблему. Вероятно, прецедент можно разбить на несколько прецедентов либо выделить альтернативные потоки.

5.7.2. Фокусируйтесь на том «что», а не «как»

Помните, прецеденты создаются для того, чтобы понять, *чего* актеры ждут от системы, а не *как* она должна это осуществлять. «Как» становится ясно позже, при проектировании. Смешение «что» с «как» является повсеместной проблемой. Разработчик, работая над прецедентом, придумывает какое-то решение. Рассмотрим, к примеру, следующий фрагмент прецедента:

```
…
4. Система просит Покупателя подтвердить заказ.
5. Покупатель нажимает кнопку OK.
…
```

В данном примере разработчик прецедента представил себе некий пользовательский интерфейс: форму с кнопкой «ОК». Из-за этого прецедент перестал быть простым изложением требований, это первичный проект. Лучше записать шаг 5 следующим образом:

```
…
5. Покупатель соглашается с заказом.
…
```

Детали проектирования (которые пока неизвестны!) должны оставаться вне прецедента.

5.7.3. Избегайте функциональной декомпозиции

> Функциональная декомпозиция не подходит для моделей прецедентов.

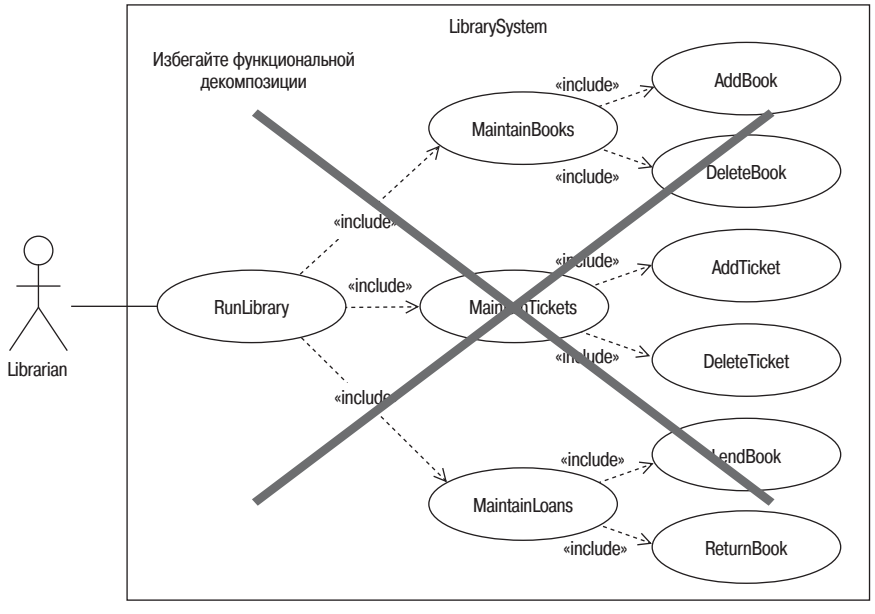


Рис. 5.16. *Пример функциональной декомпозиции*

При анализе прецедентов широко встречается следующая ошибка: создается ряд «высокоуровневых» прецедентов, которые затем разбиваются на ряд прецедентов более низкого уровня и т. д. до «элементарных» прецедентов, достаточно детализированных для реализации. Такой подход к разработке программного обеспечения называется функциональной декомпозицией. Он абсолютно *ошибочен* в применении к моделированию прецедентов.

Рассмотрим пример. На рис. 5.16 аналитик описал работу библиотечной системы с помощью одного высокоуровневого прецедента RunLibrary (управление библиотекой). Затем путем функциональной декомпозиции разложил его на все более и более детализированные уровни.

Многим не-ОО аналитикам рис. 5.16 кажется вполне приемлемым. Однако с точки зрения моделирования прецедентов в предложенном примере содержится много ошибок:

- Модель сосредоточена не на фиксировании требований, а на искусственном структурировании этих требований – существует множество возможных вариантов декомпозиции.
- Модель описывает систему как набор вложенных функций. Однако ОО системы в действительности являются наборами взаимодействующих объектов, обменивающихся сообщениями. Здесь очевидное концептуальное несоответствие.

- Интерес представляют только описания прецедентов самого низкого уровня: AddBook (добавить книгу), DeleteBook (удалить книгу), AddTicket (добавить формуляр), DeleteTicket (удалить формуляр), LendBook (выдать книгу) и ReturnBook (вернуть книгу). Более высокие уровни просто вызывают нижние и ничего не привносят в модель с точки зрения отражения требований.
- Модель сложна и непонятна заинтересованным сторонам: в ней несколько абстрактных прецедентов (RunLibrary, MaintainBooks (обслуживание книг), MaintainTickets (обслуживание формуляров) и MaintainLoans (обслуживание выдачи книг)) и много отношений «include» с более низкими уровнями абстракции.

Применение функциональной декомпозиции говорит о том, что аналитик неправильно продумал систему. Обычно это свидетельствует о том, что он обучен более традиционным методам процедурного программирования и пока что не уловил принципа ОО программирования. В этом случае лучше привлечь опытного разработчика моделей прецедентов в качестве руководителя и консультанта.

Не все примеры функциональной декомпозиции так очевидны, как приведенный на рис. 5.16. Обычно декомпозицию можно обнаружить в отдельных частях модели, поэтому желательно проверять все части модели прецедентов, имеющие глубокую иерархию.

И наконец, следует заметить, что в процессе моделирования прецедентов иерархии возникают естественным образом. Однако в этих естественных иерархиях, как правило, не более одного (или максимум двух) уровня вложенности, а модель *никогда* не строится от одного прецедента.

5.8. Что мы узнали

Мы изучили приемы углубленного моделирования прецедентов. Основная задача моделирования – создать простую понятную модель прецедентов, в которой вся необходимая информация представлена максимально четко и лаконично. Модель прецедентов, не использующая расширенные возможности, всегда предпочтительнее той, где так много обобщений, отношений «include» и «extend», что невозможно понять, о чем идет речь. Здесь лучшее правило: «Если сомневаешься, не включай».

Мы узнали следующее:

- Обобщение актеров обеспечивает возможность вынести поведение, общее для двух или более актеров, в актера-родителя.
  - Актер-родитель является более обобщенным, чем его потомки, а потомки – более специализированными, чем их родитель.
  - Дочерний актер везде может заменять актера-родителя – это принцип замещаемости.

- Актер-родитель обычно абстрактный – он определяет абстрактную роль.
- Дочерние актеры конкретны – они определяют конкретные роли.
- Обобщение актеров может упрощать диаграммы прецедентов.

- Обобщение прецедентов позволяет вынести возможности, общие для двух или более прецедентов, в родительский прецедент.
  - Дочерние прецеденты наследуют все возможности родительских прецедентов.
  - Дочерние прецеденты могут вводить новые возможности.
  - Дочерние прецеденты могут переопределять родительские возможности *за исключением* отношений и точек расширения.
  - В дочерних прецедентах мы применяем простые обозначения:
    - унаследованный без изменений – 3. (3.);
    - унаследованный и перенумерованный – 6.2. (6.1.);
    - унаследованный и переопределенный – 1. (o1.);
    - унаследованный, переопределенный и перенумерованный – 5.2. (o5.1.);
    - добавленный – 6.3.
  - Хорошим стилем является абстрактность родительского прецедента.
- Отношение «include» позволяет вынести шаги, повторяющиеся в нескольких потоках прецедентов, в отдельный прецедент, который включается по необходимости.
  - Синтаксис include(ИмяПрецедента) используется для включения поведения другого прецедента.
  - Включающий прецедент называют базовым прецедентом.
  - Прецедент, который включается, называют включаемым прецедентом.
  - Базовый прецедент без всех включаемых прецедентов является неполным.
  - Включаемые прецеденты могут быть:
    - полными – это обычные прецеденты, их экземпляры могут быть созданы;
    - неполными – они содержат только фрагмент поведения и их экземпляры не могут быть созданы.
- Отношение «extend» вводит новое поведение в существующий (базовый) прецедент.
  - Точки расширения накладываются поверх потока событий базового прецедента.
    - Точки расширения размещаются между шагами потока событий.

- Расширяющие прецеденты предоставляют сегменты вставки – фрагменты поведения, которые могут быть «подключены» к точкам расширения.
- Отношение «extend» между расширяющим и базовым прецедентами определяет точки расширения. В этих точках вводятся сегменты вставки расширяющих прецедентов.
- Базовый прецедент является полным и без сегментов вставки – он ничего не знает о возможных сегментах вставки, он только предоставляет точки входа для них.
- Расширяющий прецедент, как правило, неполный – обычно он просто объединяет в себе один или более сегментов вставки; он может быть и полным прецедентом, но это редкий случай.
- Если расширяющий прецедент имеет предусловия, они должны быть реализованы; в противном случае расширяющий прецедент не выполняется.
- Постусловия расширяющего прецедента, если они есть, накладывают ограничения на состояние системы после выполнения расширяющего прецедента.
- Расширяющий прецедент может содержать несколько сегментов вставки.
- Два или более расширяющих прецедента могут расширять один и тот же базовый прецедент в одной и той же точке расширения – порядок выполнения каждого расширяющего прецедента не определен.
- Условные расширения – логические условия, налагаемые на отношение «extend». Разрешают вставку в случае истинности условия и запрещают, если условие ложно.

- Дополнительные возможности применяются следующим образом:
  - Обобщение актеров применяется, только если упрощает модель.
  - Обобщение прецедентов рекомендуется *не* использовать или использовать только с абстрактными родителями.
  - «include» применяется, только если упрощает модель; необходимо остерегаться злоупотреблений, поскольку тогда модель прецедентов превращается в функциональную декомпозицию.
  - «extend» рекомендуется *не* использовать; если все же применяется, все разработчики модели и заинтересованные стороны должны гарантированно понимать и соглашаться с его семантикой.
- Советы и рекомендации по написанию прецедентов:
  - прецеденты должны быть короткими и простыми;
  - основное внимание необходимо уделять тому, *что*, а не *как*;
  - избегать функциональной декомпозиции.

III

Анализ

6

Рабочий поток анализа

6.1. План главы

Эта глава начинает наше исследование процесса ОО анализа. Здесь представлен краткий обзор рабочего потока анализа в UP и некоторые практические правила создания аналитических моделей, что создает базу для дальнейшего более подробного обсуждения в следующих главах этой части книги.

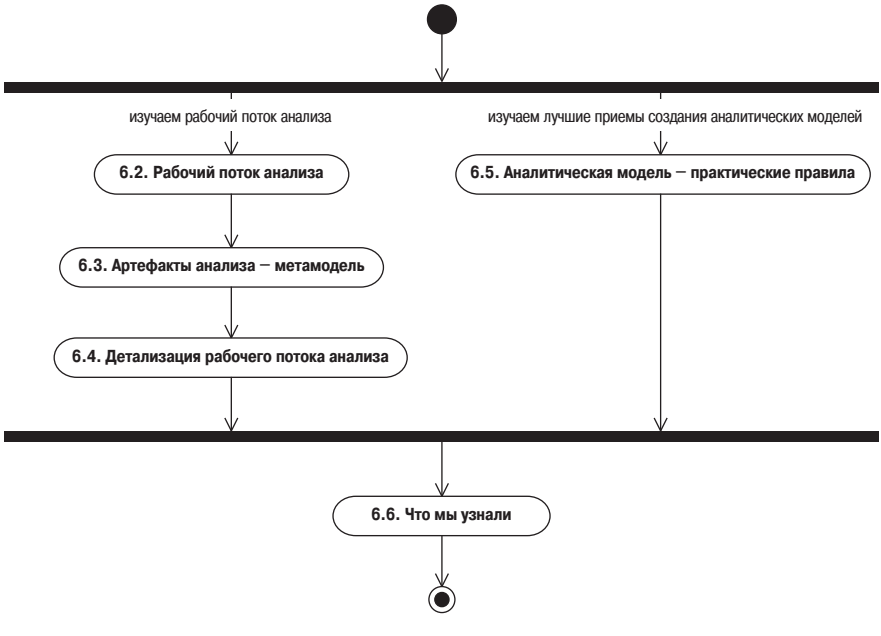


Рис. 6.1. *План главы*

6.2. Рабочий поток анализа

> Аналитическое моделирование имеет стратегически важное значение, поскольку на этом этапе делается попытка смоделировать основное поведение системы.

Основная аналитическая работа начинается в конце фазы Начало. Наряду с определением требований анализ является главной задачей фазы Уточнение.

В фазе Уточнение основные силы направлены на создание моделей, отражающих требуемое поведение системы. Как видно из рис. 6.2, анализ в большой степени пересекается с определением требований. Эти две деятельности часто идут рука об руку. Обычно необходимо провести некоторый анализ требований, чтобы сделать их более понятными и выявить все упущения или искажения.

Цель рабочего потока анализа (с точки зрения ОО анализа) – создание аналитической модели. Данная модель фокусируется на том, *что* должна делать система. Детали того, *как* система будет это делать, предоставляются потоку проектирования.

Граница между анализом и проектированием довольно неопределенная, и до известной степени все зависит от аналитика. В разделе 6.5 представлены некоторые практические правила, которые могут помочь в создании хороших аналитических моделей.

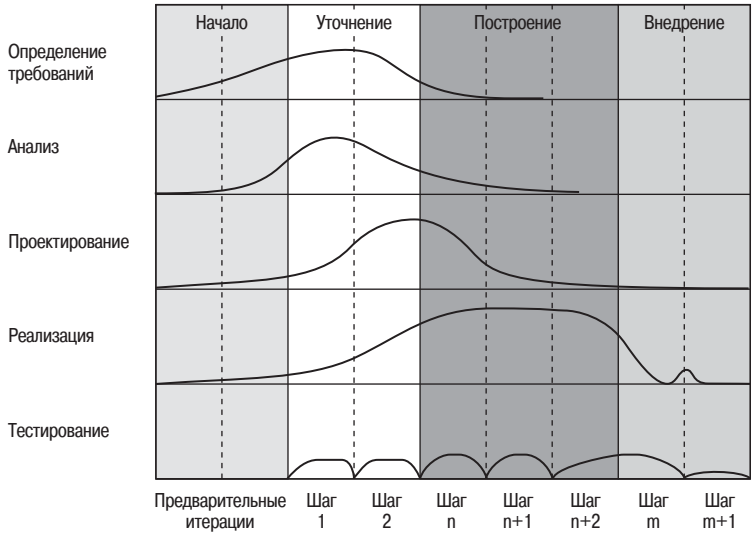


Рис. 6.2. *Определение требований выполняется в фазах Начало и Уточнение. Адаптировано с рис. 1.5 [Jacobson 1] с разрешения издательства Addison-Wesley*

6.3. Артефакты анализа – метамодель

В рабочем потоке анализа создаются два ключевых артефакта:

- классы анализа – ключевые понятия в бизнес-сфере;
- реализации прецедентов – иллюстрируют, как экземпляры классов анализа могут взаимодействовать для реализации поведения системы, описанного прецедентами.

С помощью UML можно самостоятельно создавать аналитическую модель. Метамодель аналитической модели приведена на рис. 6.3.

Синтаксис пакета был представлен ранее (сущности, имеющие вид папок). Синтаксис класса (прямоугольники) и синтаксис реализации прецедента (овалы, нарисованные пунктирной линией) показаны здесь впервые. Классы рассматриваются в главе 7, пакеты – в главе 11, реализации прецедентов – в главе 12.

Аналитическую модель можно представить в виде пакета с треугольником в верхнем правом углу. Этот пакет содержит один или более пакетов анализа. Мы называем их «пакетами анализа», потому что они являются частью аналитической модели. На рис. 6.3 показаны лишь четыре пакета анализа, но настоящие аналитические модели могут включать множество пакетов. В каждом пакете, в свою очередь, могут находиться вложенные пакеты анализа.

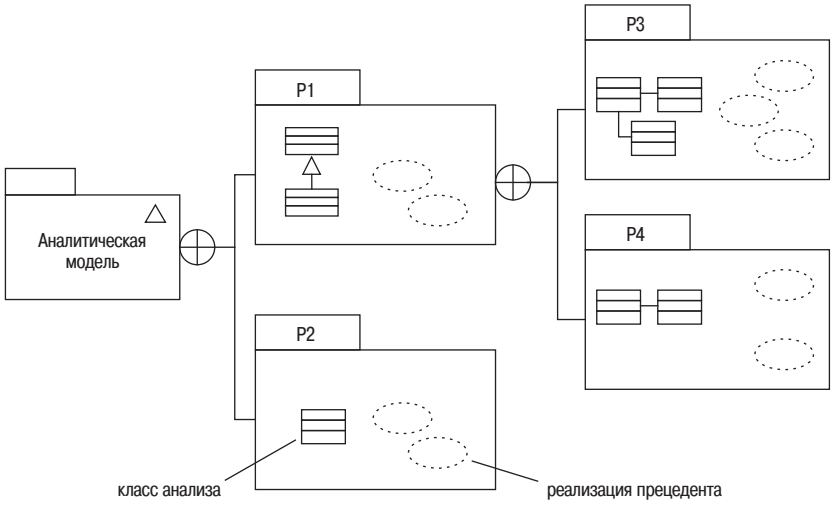


Рис. 6.3. *Метамодель аналитической модели*

6.4. Детализация рабочего потока анализа

На рис. 6.4 показан рабочий поток анализа в UP. Соответствующие деятельности будут рассмотрены в следующих главах. Но чтобы по-

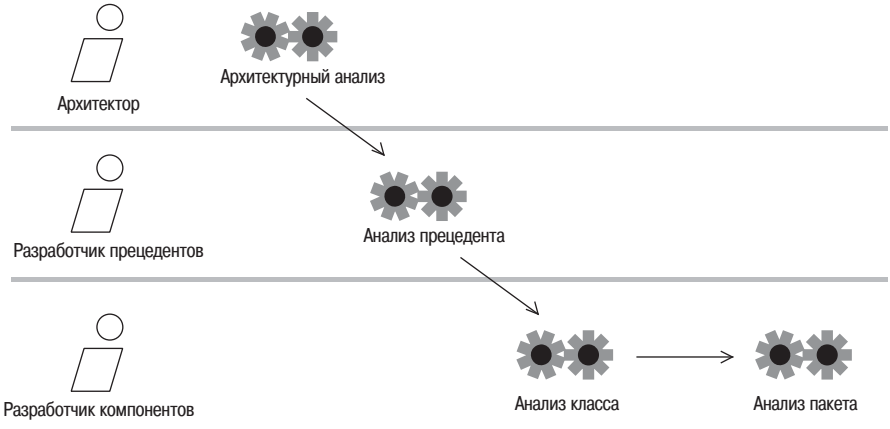


Рис. 6.4. *Рабочий поток анализа в UP. Воспроизведено с рис. 8.18 [Jacobson 1] с разрешения издательства Addison-Wesley*

нять их, необходимо рассмотреть классы и объекты. Они будут обсуждаться в главе 7.

6.5. Аналитическая модель – практические правила

Все системы разные, поэтому сложно найти универсальный подход к созданию аналитических моделей. Тем не менее аналитическая модель системы среднего размера и сложности включает от 50 до 100 классов анализа. Необходимо запомнить, что при создании аналитической модели крайне важно ограничиться *лишь* теми классами, которые являются частью словаря предметной области. Всегда есть искушение поместить в аналитическую модель и проектные классы (такие как классы, используемые для установления соединений или доступа к базе данных), но этого необходимо избегать (если только предметная область не касается связи или баз данных)! Такое ограничение накладывается в попытке сделать аналитическую модель кратким, простым описанием структуры и поведения системы. Все решения по реализации должны приниматься в рабочих потоках проектирования и реализации.

Вот некоторые практические приемы создания хорошей аналитической модели:

- Аналитическая модель *всегда* создается на языке соответствующей сферы деятельности. Абстракции аналитической модели должны формировать часть словаря бизнес-сферы.
- Создаваемые модели должны «рассказывать историю». Каждая диаграмма должна раскрывать некоторые важные части поведения системы. Иначе зачем они нужны? Как создавать такие диаграммы, будет показано при рассмотрении реализации прецедентов.

- Необходимо сосредоточиться на отображении основной картины. Не надо углубляться в детали того, как будет работать система. Для этого отводится достаточно времени при проектировании.
- Необходимо четко различать предметную область (бизнес-требования) и область решения (детальные конструктивные соображения). Основное внимание всегда должно быть направлено на абстракции предметной области. Например, при моделировании системы электронной торговли в аналитической модели должны присутствовать классы Customer (клиент), Order (заказ) и ShoppingBasket (корзина покупок). Здесь *не* должно быть классов доступа к базе данных или классов для установления соединений, поскольку они – артефакты области решения.
- Всегда необходимо стремиться минимизировать связанность (coupling). Каждая ассоциация между классами увеличивает их связанность. В главе 9 будет показано, как можно максимально сократить связанность с помощью кратностей и навигации по ассоциациям.
- *Если* присутствует естественная и очевидная иерархия абстракций, должна быть рассмотрена возможность применения наследования. При анализе иерархия никогда не должна применяться просто для повторного использования кода. Как мы увидим в разделе 17.6, наследование – самая сильная форма связанности классов.
- Всегда должен задаваться вопрос: «Модель полезна для всех заинтересованных сторон?» Нет ничего хуже, чем создавать аналитическую модель, которая игнорируется пользователями или проектировщиками и разработчиками. Пока что это происходит очень часто, особенно с неопытными аналитиками. Основная превентивная стратегия при этом – сделать аналитическую модель и процесс моделирования максимально открытыми, по возможности привлечь в него заинтересованные стороны, проводить частые и публичные обзоры.

И наконец, модель должна быть простой! Конечно, легче сказать, чем сделать, но нам из собственного опыта известно, что внутри любой сложной аналитической модели можно найти простую. Один из способов упрощения – рассматривать вопрос в общем, не погружаясь в частности.

6.6. Что мы узнали

Мы рассмотрели следующее:

- Анализ заключается в создании моделей, отображающих основные требования и характеристики целевой системы – аналитическое моделирование имеет стратегическое значение.
- Основной объем работ рабочего потока анализа выполняется в конце фазы Начало и в фазе Уточнение.

- Рабочие потоки анализа и определения требований пересекаются, особенно в фазе Уточнение – обычно для выявления неучтенных или искаженных требований полезно анализировать требования по мере их обнаружения.
- Аналитическая модель:
  - всегда создается на языке соответствующей сферы деятельности;
  - отображает картину в целом;
  - содержит артефакты, моделирующие предметную область;
  - рассказывает историю о целевой системе;
  - полезна максимально возможному числу заинтересованных сторон.
- Аналитические артефакты – это:
  - классы анализа – ключевые понятия бизнес-сферы;
  - реализации прецедентов – иллюстрируют, как экземпляры классов анализа могут взаимодействовать для реализации поведения системы, описанного прецедентами.
- Рабочий поток анализа в UP включает следующие деятельности:
  - Архитектурный анализ
  - Анализ прецедента
  - Анализ класса
  - Анализ пакета
- Аналитическая модель – практические правила:
  - аналитическая модель среднестатистической системы состоит из 50–100 классов анализа;
  - включаются только классы, моделирующие словарь предметной области;
  - *не* формируются решения по реализации;
  - основное внимание на классах и ассоциациях – минимизация связанности между классами;
  - наследование применяется там, где присутствует естественная иерархия абстракций;
  - необходимо стремиться к простоте модели!

7

Объекты и классы

7.1. План главы

Эта глава полностью посвящена объектам и классам. Они – основные строительные блоки ОО систем. Если читатель уже хорошо знаком с понятием объектов и классов, можно пропустить разделы 7.2 и 7.4. Однако, вероятно, вам будет интересна информация о UML-нотации объектов (раздел 7.3) и классов (раздел 7.5).

Глава завершается обсуждением смежных вопросов области действия операций и атрибутов (раздел 7.5), а также вопросов создания и уничтожения объектов (раздел 7.7).

При анализе используется только часть UML-синтаксиса класса. НО чтобы не разбрасывать справочную информацию по всей книге и не возвращаться к этому позже, в этой главе рассматривается полный синтаксис класса.

7.2. Что такое объекты?

Книга «UML Reference Manual» [Rumbaugh 1] определяет объект как «отдельную сущность с явно выраженными границами, которая инкапсулирует состояние и поведение; экземпляр класса».

> Объекты объединяют данные и функциональность в единый блок.

Объект можно представить как единый пакет данных и функциональности. Как правило, единственный путь добраться до данных объекта – вызвать одну из предоставляемых им функций. Эти функции называются *операциями* (*operations*). Сокрытие данных объекта за уровнем операций известно как инкапсуляция (encapsulation), или сокрытие данных (data-hiding). Инкапсуляция в UML не является обязательной, поскольку некоторые ОО языки не нуждаются в ней. Однако со-

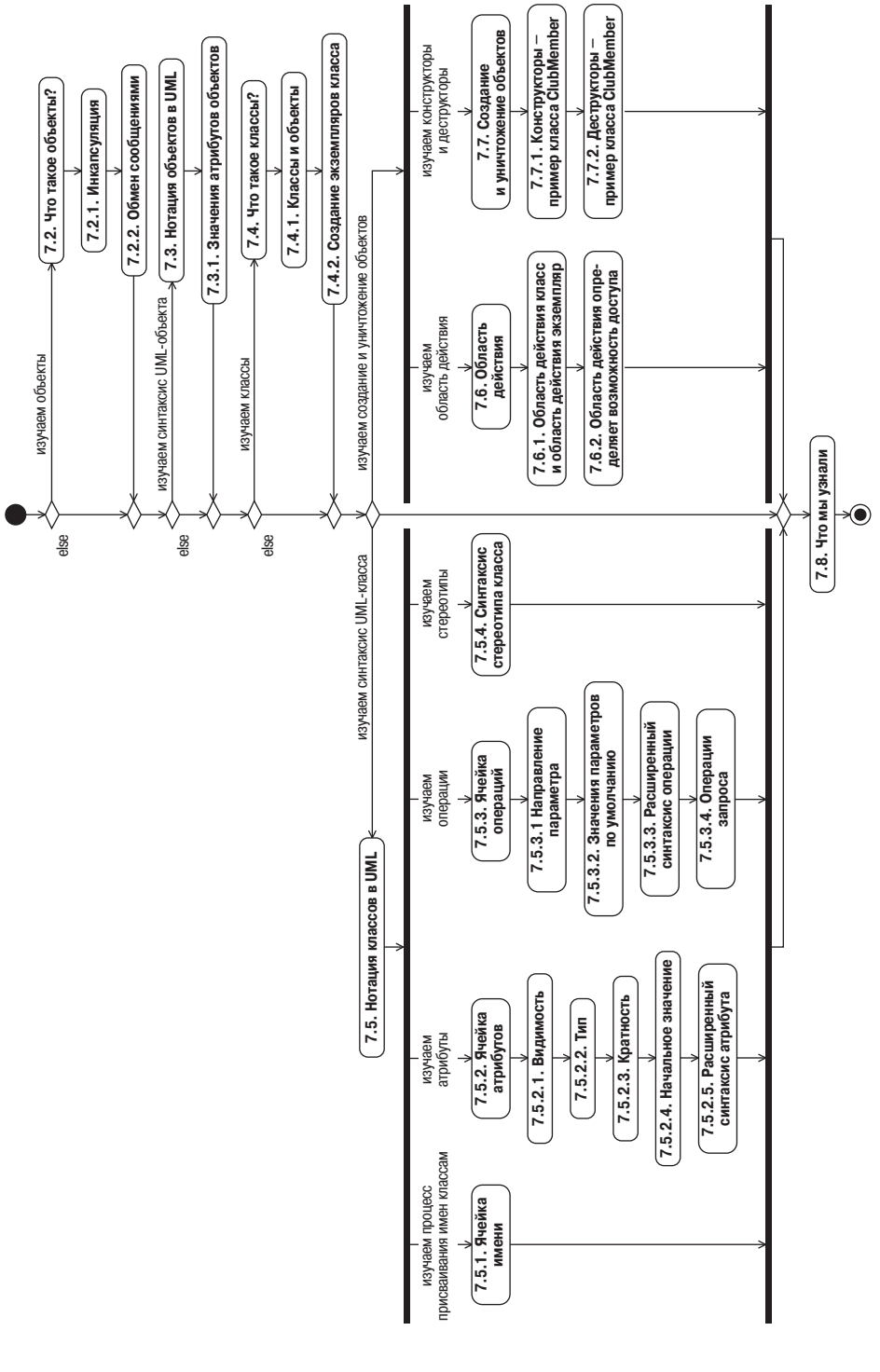


Рис. 7.1. План главы

крытие данных объекта за уровнем операций всегда считается хорошим ОО стилем.

> Объекты скрывают данные на уровне функций, которые называются операциями.

Каждый объект является экземпляром некоторого класса, определяющего общий набор свойств (атрибутов и операций), присущих всем экземплярам этого класса. Идея классов и классификаторов на самом деле очень проста. Представим принтер типа «Epson Photo 1200». Он описывает свойства всех отдельных экземпляров данного класса, в том числе и конкретный «Epson Photo 1200 с/н 34120098», стоящий на нашем столе. Конкретный экземпляр класса называется объектом.

Немного поразмыслив над этим примером объекта принтера Epson, можно увидеть, что ему присущи определенные свойства, общие для всех объектов.

> Каждый объект имеет уникальный идентификатор.

- Идентификатор (identity) – это определение существования и единственности объекта во времени и пространстве. Это то, что отличает его от всех остальных объектов. В нашем примере серийный номер может использоваться в качестве идентификатора для обозначения конкретного принтера на нашем столе и представления уникального идентификатора этого объекта. Серийный номер – замечательный способ идентифицировать физический объект. Для идентификации каждого программного объекта, принимающего участие в ОО анализе и проектировании, используется аналогичный принцип – идея объектной ссылки. Конечно, в реальности не у всех объектов есть серийный номер, но все равно они имеют уникальные идентификаторы: конкретные пространственные и временные координаты. Подобным образом в ОО программных системах каждый объект имеет некоторую объектную ссылку.

> Значения атрибутов хранят данные объекта.

- Состояние (state) – определяется значениями атрибутов объекта и его отношениями с другими объектами в конкретный момент времени. В табл. 7.1. приведен полный список возможных состояний принтера, из которого видно, как состояние объекта зависит от значений его атрибутов и его связи с другими объектами.
- Поведение (behavior) – принтер может выполнять конкретные действия:

  switchOn() (включиться)
  switchOff() (выключиться)

printDocument() (распечатать документ)
pageFeed() (заправить бумагу)
clearInkJetNozzles() (очистить форсунки)
changeInkCartridge() (заменить картридж)

Вызов операции объекта всегда приводит к изменению значений одного или более его атрибутов или отношений с другими объектами. Это *может* обусловить переход состояний – целенаправленный переход объекта из одного состояния в другое. Из табл. 7.1 видно, что состояние объекта может влиять и на его поведение. Например, если в принтере закончились чернила (состояние объекта = OutOfBlackInk), вызов операции printDocument() приведет к сообщению об ошибке. Поэтому поведение printDocument() является *зависимым от состояния*.

> Метод – это реализация операции.

Операция – это описание части поведения. Реализация этого поведения называется *методом* (*method*).

Таблица 7.1

Состояние объекта	Атрибут класса	Значение атрибута объекта	Отношение
On	power	on	Не определено
Off	power	off	Не определено
OutOfBlackInk	blackInkCartridge	empty	Не определено
OutOfColorInk	colorInkCartridge	empty	Не определено
Connected	не определено	не определено	Подключен к объекту компьютер
NotConnected	не определено	не определено	Не подключен к объекту компьютер

7.2.1. Инкапсуляция

Как уже говорилось, идентификатор объекта – это некий уникальный дескриптор, обычно адрес памяти, предоставляемый языком реализации. С этого момента будем называть эти дескрипторы объектными ссылками. В ОО анализе не надо беспокоиться об их реализации. Можно просто принять, что у каждого объекта есть уникальный идентификатор, управляемый технологией реализации. Возможно, при проектировании понадобится рассмотреть реализацию объектных ссылок. Это необходимо, если целевым языком является ОО язык программирования, такой как C++, который позволяет напрямую работать с определенными типами объектных ссылок, известными как указатели.

На рис. 7.2 приведено концептуальное представление объекта, подчеркивающее инкапсуляцию. Обратите внимание, что рис. 7.2 *не яв-*

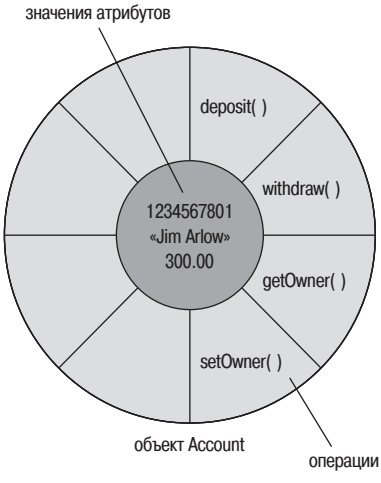


Рис. 7.2. *Представление объекта, подчеркивающее инкапсуляцию*

ляется UML-диаграммой. UML-синтаксис для объектов будет показан позже.

> Состояние объекта определяется значениями его атрибутов.

Состояние объекта – это набор значений атрибутов (в данном случае 1234567801, «Jim Arlow», 300.00) объекта в любой момент времени. Значения некоторых атрибутов постоянны, значения других могут со временем меняться. Например, номер счета и имя останутся постоянными, а вот баланс, надеемся, будет постепенно увеличиваться!

Поскольку баланс меняется со временем, мы видим, что состояние объекта также изменяется во времени. Например, если баланс отрицательный, можно сказать, что объект находится в состоянии Overdrawn (кредит превышен). При изменении баланса из отрицательного в нулевой объект существенно меняет свою суть: переходит из состояния Overdrawn в состояние Empty (пустой). Более того, когда баланс объекта Account (счет) становится положительным, осуществляется еще один переход состояния: из Empty в InCredit (кредитоспособен). Возможны и другие переходы состояний. Фактически любой вызов операции, приводящий к изменению сути объекта, обуславливает переход состояний. UML предоставляет мощный набор методов моделирования изменений состояния, которые называются конечными автоматами; им посвящена глава 21.

> Поведение объекта – это то, «что он может сделать для нас», т. е. его операции.

Поведение любого объекта – это, по существу, то, «что он может сделать» для вас. Объект на рис. 7.2 предоставляет операции, перечисленные в табл. 7.2.

Таблица 7.2

Операция	Семантика
deposit()	Размещает некоторую сумму в объекте Account. Увеличивает значение атрибута balance.
withdraw()	Снимает некоторую сумму с Account. Уменьшает значение атрибута balance.
getOwner()	Возвращает владельца объекта Account – операция запроса.
setOwner()	Меняет владельца объекта Account.

Этот набор операций определяет поведение объекта. Обратите внимание, что вызов некоторых из этих операций (deposit(), withdraw(), setOwner()) приводит к изменению значений атрибута и *может* генерировать переходы состояния. Операция getOwner() не меняет значения атрибута и поэтому *не* приводит к переходу состояния.

Инкапсуляция, или сокрытие данных, – одно из основных преимуществ ОО программирования. Она обеспечивает возможность создания более надежного и расширяемого программного обеспечения. В этом простом примере объекта Account пользователю не надо беспокоиться о структуре данных, сокрытых в объекте. Его интересует только то, что объект может сделать. Иначе говоря, ему интересны *сервисы* (*services*), предлагаемые другим объектам.

7.2.2. Обмен сообщениями

> Объекты формируют поведение системы путем обмена сообщениями по связям. Это – кооперация.

У объектов есть значения атрибутов и поведение, но как эти объекты объединить, чтобы создать систему программного обеспечения? Объекты кооперируются для осуществления функций системы. Это означает, что они устанавливают связи с другими объектами и обмениваются сообщениями по этим связям. Когда объект получает сообщение, он проверяет набор своих операций в поиске той, сигнатура которой соответствует сигнатуре сообщения. Если таковая имеется, он инициирует эту операцию (рис. 7.3). В сигнатуру входят имя сообщения (или операции), типы параметров и возвращаемое значение.

Во время выполнения ОО система состоит из множества создаваемых объектов, существующих некоторое время и затем, возможно, уничтожаемых. Эти объекты обмениваются сообщениями, инициируя сервисы друг друга. Такая структура радикально отличается от процедур-

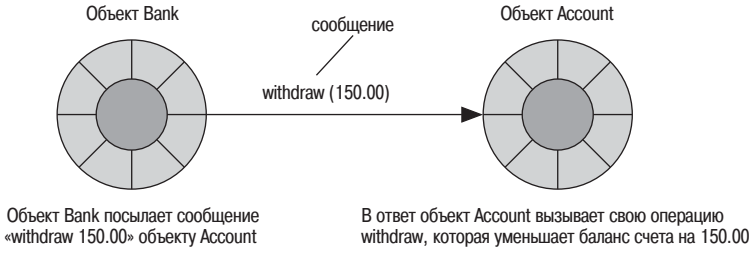


***Рис. 7.3.** Обмен сообщениями*

ных программных систем, которые изменяются со временем путем последовательного применения функций к данным.

7.3. Нотация объектов в UML

Пиктограмма объекта в UML – это прямоугольник с двумя ячейками (рис. 7.4). В верхней ячейке размещается идентификатор объекта, который *всегда* подчеркивается. Это важно, поскольку в UML обозначения классов и объектов очень похожи. Если строго следовать правилам применения подчеркивания, никогда не возникнет вопроса, чем является моделируемый элемент – классом или объектом.

UML очень гибок относительно представления объектов на диаграммах объектов. Идентификатор объекта может включать следующие элементы.

- Только имя класса, например :Account. Это означает, что имеется анонимный объект или экземпляр данного класса (т. е. это экземпляр класса Account, но он не идентифицирован или в действительности не имеет значения, какой именно это экземпляр). Анонимные объекты обычно используются, когда на данной диаграмме присутствует только один объект этого конкретного класса. Если необходимо показать два объекта одного и того же класса, каждому из них должно быть присвоено уникальное имя, чтобы можно было их различать.

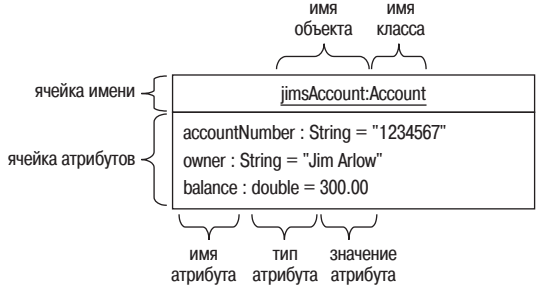


***Рис. 7.4.** Нотация объектов в UML*

- Только имя объекта, например jimsAccount. Здесь обозначен конкретный объект, но не указано, какому классу он принадлежит. Данное обозначение может быть полезным на самых ранних стадиях анализа, когда еще не выявлены все классы.
- Если указываются и имя объекта, и имя класса, они разделяются двоеточием. Двоеточие может читаться как «является экземпляром класса». Таким образом, диаграмму на рис. 7.4 можно прочитать так: существует объект с именем jimsAccount, который является экземпляром класса Account.

> Объекты одного класса имеют одинаковые операции и атрибуты, но значения их атрибутов могут быть разными.

Имена объектов обычно записываются заглавными и строчными буквами вперемежку, начиная со строчной буквы. Следует избегать специальных символов, таких как пробелы и подчеркивания. Такой стиль записи называют *lowerCamelCase*, потому что в результате получаются «горбатые» на вид слова.

Как сказано в разделе 7.2, класс определяет атрибуты и операции набора объектов. Поскольку все объекты *одного* класса имеют совершенно *одинаковый* набор операций, они перечисляются в пиктограмме класса, а не в пиктограмме объекта.

Атрибуты по выбору могут быть приведены в нижней ячейке пиктограммы объекта. Тем атрибутам, которые решено вынести на диаграмму, *должны* быть присвоены имена. Их тип и значение указывать не обязательно. Имена атрибутов также записываются в стиле lowerCamelCase.

7.3.1. Значения атрибутов объектов

Значение каждого атрибута записывается следующим образом:

```
имя : тип = значение
```

Можно отображать все, некоторые или вообще не отображать значения атрибутов. Все зависит от назначения диаграммы.

Чтобы сохранить простоту и ясность диаграммы, можно опускать типы атрибутов, поскольку они уже определены в классе объекта. Когда будет показано применение диаграмм объектов при анализе (глава 12), станет понятно, почему может быть принято решение отображать в пиктограмме объекта не всю информацию.

7.4. Что такое классы?

Книга «UML Reference Manual» [Rumbaugh 1] определяет класс как «дескриптор набора объектов, имеющих одинаковые атрибуты, операции,

методы, отношения и поведение». Подытожить это можно так: класс – это дескриптор набора объектов, имеющих одинаковые свойства.

> Класс описывает свойства ряда объектов.

Каждый объект – это экземпляр только одного класса. Вот несколько рекомендаций относительно классов.

- Класс надо рассматривать как шаблон объектов: класс определяет структуру (набор свойств) всех объектов этого класса. Все объекты *одного* класса должны иметь *одинаковый* набор операций, *одинаковый* набор атрибутов и *одинаковый* набор отношений, но значения атрибутов могут быть *различными*.
- Класс – это штамп, а объекты – отпечатки этого штампа на листке бумаги. Или класс – это форма для печенья, а объекты – печенье.

> Каждый объект – это экземпляр только одного класса.

Классификатор и экземпляр – принятые деления UML (см. главу 1), и самым обычным примером этого деления являются класс и объект. Класс – это спецификация или шаблон, которому должны следовать все объекты этого класса (экземпляры). Атрибуты, описанные классом, в каждом объекте имеют конкретные значения. Каждый объект будет отвечать на сообщения, инициируя операции, описанные классом.

В зависимости от их состояния разные объекты могут отвечать на одно и то же сообщение по-разному. Например, попытка снять $100 с банковского счета, на котором уже превышен кредит, приведет к результату, отличному от того, если попытаться снять $100 со счета, на котором есть несколько сотен долларов кредита.

Классификация – это, наверное, единственный имеющийся у людей способ упорядочить информацию о мире. По существу, это также одна из самых важных ОО концепций. Применяя понятие классов, можно говорить об отдельном типе машин или о виде дерева, даже не упоминая конкретный экземпляр. То же самое в программном обеспечении. Классы позволяют описывать набор свойств, которыми *должен* обладать каждый объект класса, без необходимости описывать каждый из этих объектов.

Посмотрим на рис. 7.5 и подумаем, сколько классов изображено на этом рисунке?

На самом деле ответа на этот вопрос нет! Существует практически неисчислимое количество способов классификации объектов реального мира. Вот несколько классов, которые можно увидеть:

- класс кошек;
- класс жирных прожорливых котов (у нас есть кот – экземпляр этого класса!);

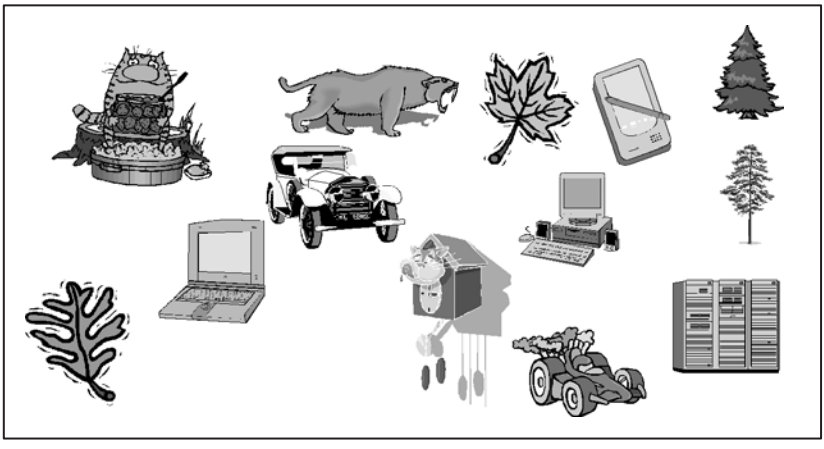


Рис. 7.5. *Сколько классов здесь изображено?*

- класс деревьев;
- класс листьев;
- и т. д., и т. п.

> Правильный выбор схемы классификации – один из ключей к успешному ОО анализу.

Поскольку существует такое огромное количество вариантов, выбор наиболее подходящей схемы классификации – один из самых важных аспектов ОО анализа и проектирования. Как это делать, будет показано в главе 8.

Посмотрев на рис. 7.5 внимательнее, можно увидеть другие типы отношений кроме уже упоминавшегося класс/экземпляр. Например, можно обнаружить несколько уровней классификации. Есть класс кошек. Классификацию можно расширить и выделить подклассы «домашних кошек» и «диких кошек». Или даже подклассы «современные кошки» и «доисторические кошки». Это отношение между классами: один класс является подклассом другого. И наоборот, класс «кошки» является суперклассом «домашних кошек» и «диких кошек». Более подробно об этом рассказывается в главе 10, посвященной наследованию.

На рис. 7.5 видно, что у объектов «деревья» есть наборы объектов «листья». Это очень прочный вид отношений между деревьями и листьями. Каждый объект «лист» принадлежит определенному объекту «дерево». Деревья не могут обмениваться или совместно использовать листья. И жизненный цикл листа тесно связан и контролируется деревом. Такое отношение между объектами в UML называют композицией (composition).

Однако отношения между, скажем, компьютерами и внешними устройствами совершенно другие. Компьютеры могут обмениваться внешними устройствами, например такими, как пара динамиков. Разные компьютеры даже могут совместно использовать некоторые устройства. Кроме того, если компьютер выходит из строя, его внешние устройства могут прекрасно пережить его и использоваться новой машиной. Жизненный цикл внешних устройств обычно не зависит от жизненного цикла компьютера. В UML этот тип отношений объектов называется агрегацией (aggregation). Отношения объектов, в частности композиция и агрегация, более подробно рассматриваются в главе 18.

7.4.1. Классы и объекты

> Отношения объединяют сущности.

Между классом и объектами этого класса устанавливается отношение «instantiate» (создать экземпляр). Это первый встречающийся нам пример отношения. Книга «UML Reference Manual» [Rumbaugh 1] определяет отношение как «связь между элементами модели». В UML существует множество типов отношений, и со временем все они будут нами рассмотрены.

> Отношение зависимости означает, что изменение сущности-поставщика оказывает влияние на сущность-клиент.

Отношение «instantiate» между объектами и классами показано на рис. 7.6. Пунктирная линия со стрелкой обозначает отношение зависимости, которому стереотип «instantiate» придает особое значение. Как было показано в главе 1, все, что находится во французских кавычках («...»), называется стереотипом. Стереотипы – один из трех механиз-

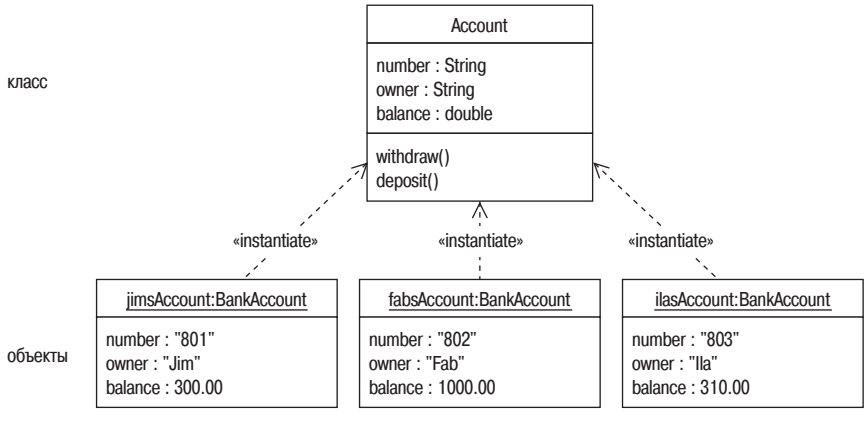


***Рис. 7.6.** Отношение «instantiate»*

мов расширения в UML. Стереотип – это способ настройки элементов модели, способ создания вариантов с новой семантикой. В данном случае стереотип «instantiate» превращает обычную зависимость в отношение конкретизации между классом и объектами этого класса.

«UML Reference Manual» [Rumbaugh 1] определяет зависимость как «отношение между двумя элементами, при котором изменение одного элемента (поставщика) может влиять или поставлять информацию, необходимую другому элементу (клиенту)». Из рис. 7.6 очевидно, что класс Account должен быть поставщиком, потому что он определяет структуру всех объектов этого класса, а объекты – это клиенты.

7.4.2. Создание экземпляров класса

Создание экземпляров (instantiation) – это создание новых экземпляров элементов модели. В данном случае создаются объекты классов. Создаются *новые экземпляры* классов.

> При создании экземпляра класса создается новый объект. Класс используется как шаблон.

UML старается быть универсальным, поэтому создание экземпляров применяется не только к классам и объектам, но и к другим элементам модели. Понятие создания экземпляров, по сути, представляет общий процесс создания конкретного экземпляра чего-либо по шаблону.

В большинстве ОО языков есть специальные операции, называемые конструкторами, которые на самом деле принадлежат классу, а не объектам этого класса. Говорят, что область действия этих специальных операций – класс. Более подробно на области действия мы остановимся в разделе 7.6. Назначение операции конструктор – создание новых экземпляров класса. Конструктор выделяет память для нового объекта, присваивает ему уникальный идентификатор и задает исходные значения атрибутов. Он также настраивает все связи с другими объектами.

7.5. Нотация классов в UML

Визуальный синтаксис UML для класса очень богат. Чтобы синтаксис был управляемым, в UML существует понятие необязательных дополнений. Обязательной частью в визуальном синтаксисе является только ячейка с именем класса. Все остальные ячейки и дополнения необязательны. Однако для справки на рис. 7.7 показано все.

> Показывайте только важные ячейки и дополнения.

Какие ячейки и дополнения будут включены в класс на диаграмме классов, зависит только от назначения диаграммы. Если интерес представляют лишь отношения между различными классами, тогда может

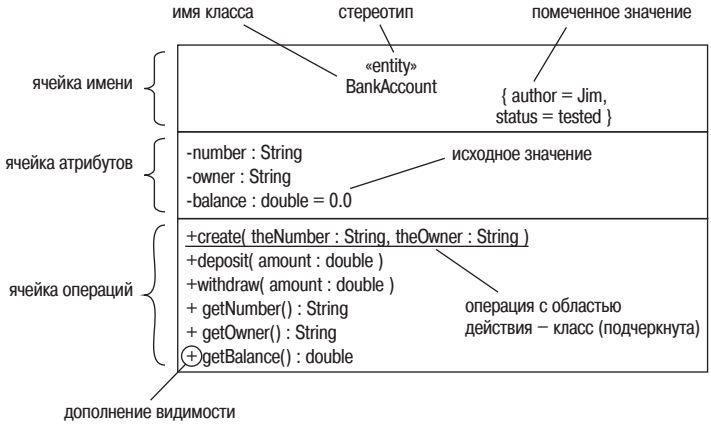


Рис. 7.7. *Нотация классов в UML*

быть достаточно ячейки с именем. Если диаграмма призвана проиллюстрировать поведение классов, тогда, вероятно, в каждый класс будет добавлена ячейка с ключевыми операциями. С другой стороны, диаграмма может быть более «ориентированной на данные», например представлять проецирование классов в реляционные таблицы. В этом случае должны быть показаны ячейки имени и атрибутов, возможно, и типы атрибутов. Необходимо стремиться использовать эту гибкость UML и размещать на диаграмме классов именно такой объем информации, который необходим для четкого и лаконичного представления идеи.

В аналитических моделях обычно необходимо показывать только:

- имя класса;
- ключевые атрибуты;
- ключевые операции;
- стереотипы (если они приносят пользу делу).

Обычно *не* показывают следующее:

- помеченные значения;
- параметры операций;
- видимость;
- исходные значения (если только они не значимы для дела).

В нескольких следующих подразделах будут более подробно рассмотрены ячейки имени, атрибутов и операций.

7.5.1. Ячейка имени

Хотя UML не определяет для классов никаких обязательных соглашений о присваивании имен, существует общепринятое соглашение.

- Имя класса записывается в стиле UpperCamelCase: имя и все слова, его образующие, пишутся с заглавной буквы. Специальные символы, такие как знаки препинания, тире, подчеркивание, «&», «#» и слэш, *никогда* не используются. Для этого есть достаточное основание: эти символы применяются в таких языках, как HTML и XML, и в операционных системах. Поэтому их применение в именах классов или именах любых других элементов модели может привести к неожиданным последствиям при генерации из модели кода или документации HTML/XML.

> Никогда не используйте аббревиатуры в именах классов, атрибутов или операций.

- *Во что бы то ни стало* необходимо избегать сокращений. Имена классов всегда должны отражать имена реальных сущностей *без* сокращений. Например, имя FlightSegment всегда более предпочтительно, чем FltSgmnt, DepositAccount предпочтительнее, чем DpstAccnt. Опять же причина этому очень простая – аббревиатуры затрудняют чтение модели (и результирующего кода). Время, сэкономленное при наборе, не сравнимо со временем, необходимым для обслуживания моделей или кода с сокращенными именами.
- Если есть специальные акронимы данной предметной области (например, CRM (Customer Relationship Management – управление взаимоотношениями с клиентами)), широко используемые и понятные *всем* пользователям модели, они могут применяться. Однако необходимо помнить, что полное имя, если оно делает модель более понятной, все равно предпочтительнее.

Классы представляют «сущности», поэтому их имена должны быть существительными или именной группой, например Person (человек), Money (деньги), BankAccount (банковский счет).

7.5.2. Ячейка атрибутов

Единственной обязательной частью UML-синтаксиса для атрибутов (рис. 7.8) является имя атрибута. Имена атрибутов записываются в стиле lowerCamelCase, т. е. начинаются со строчной буквы, а далее большие и малые буквы пишутся вперемежку. Обычно в качестве имен атрибутов используются имена существительные или именные группы, потому что атрибуты указывают на некоторую «сущность», например баланс счета. Необходимо избегать специальных символов и сокращений.

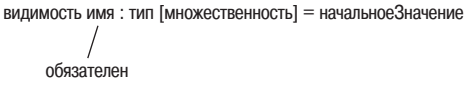


***Рис. 7.8.** UML-синтаксис для атрибутов*

Необязательные части синтаксиса атрибутов рассматриваются в следующих нескольких разделах.

7.5.2.1. Видимость

> Видимость контролирует доступ к свойствам класса.

Дополнение видимости (visibility) (табл. 7.3) применяется к атрибутам и операциям класса. Оно также может применяться к именам ролей в отношениях (глава 9). При анализе диаграммы обычно не загромождают видимостями, поскольку, по сути, это дополнение дает описание того «как», а не «что».

Таблица 7.3

Дополнение	Видимость	Семантика
+	Public (Открытый)	Любой элемент, который имеет доступ к классу, имеет доступ к любой из его возможностей, видимость которой public.
–	Private (Закрытый)	Только операции класса имеют доступ к возможностям, имеющим видимость private.
#	Protected (Защищенный)	Только операции класса или потомка класса имеют доступ к возможностям, имеющим видимость protected.
~	Package (Пакетный)	Любой элемент, находящийся в одном пакете с классом или во вложенном подпакете, имеет доступ ко всем его возможностям, видимость которых package.

Языки реализации могут по-разному интерпретировать все типы видимости UML, кроме public (открытый). Это важно. Фактически языки программирования могут определять дополнительные типы видимости, которые UML не поддерживает по умолчанию. Обычно это не проблема. Самые распространенные ОО языки программирования, такие как C++ и Java, и даже популярные полуОО языки, например Visual Basic, замечательно справляются с видимостями public, private (закрытый), protected (защищенный) и package (пакетный), по крайней мере, в первом приближении.

Давайте более подробно рассмотрим два ОО языка. В табл. 7.4 проводится сравнение семантики видимости UML и языков программирования Java и C#.

Видимость зависит от языка реализации и может стать очень сложной. Какой именно тип видимости используется, является конкрет-

ным решением реализации, которое обычно принимается программистом, а не аналитиком/проектировщиком. Для общего моделирования стандартных UML-определений public, private, protected и package вполне достаточно. Рекомендуем ограничиться ими.

Таблица 7.4

Видимость	Семантика UML	Семантика Java	Семантика C#
public	Любой элемент, который имеет доступ к классу, имеет доступ к любой из его возможностей, видимость которой public.	Аналогично UML.	Аналогично UML.
private	Только операции класса имеют доступ к возможностям, имеющим видимость private.	Аналогично UML.	Аналогично UML.
protected	Только операции класса или потомка класса имеют доступ к возможностям, имеющим видимость protected.	Аналогично UML, но доступ также имеют все классы, находящиеся в том же пакете Java, что и определяющий класс.	Аналогично UML.
package	Любой элемент, находящийся в одном пакете с классом или во вложенном подпакете, имеет доступ ко всем его возможностям, видимость которых package.	Применяемая по умолчанию видимость в Java: вложенным классам вложенных подпакетов доступ к элементам родительского пакета автоматически не предоставляется.	–
private protected (закрытый защищенный)	–	Аналогично protected в UML.	–
internal (внутренний)	–	–	Доступны любому элементу той же программы.
protected internal (защищенный внутренний)	–	–	Сочетает семантику protected и internal, применимо только к атрибутам.

7.5.2.2. Тип

> Типом атрибута может быть другой класс или примитивный тип.

Типом атрибута может быть другой класс или примитивный тип.

Спецификация UML определяет четыре примитивных типа, используемых в самой спецификации UML. Их можно применять в аналитических моделях, если необходимо сохранить платформонезависимость. Эти типы приведены в табл. 7.5.

Таблица 7.5

	Простой тип	Семантика
UML	Integer	Целое число.
	UnlimitedNatural	Целое число >= 0. Бесконечность обозначается как *.
	Boolean	Может принимать значения true или false.
	String	Последовательность символов. Строковые литералы заключаются в кавычки, например "Джим".
OCL	Real	Число с плавающей точкой.

Объектный язык ограничений (Object Constraint Language, OCL) – формальный язык для выражения ограничений в UML-моделях. Подробно OCL обсуждается в главе 25. OCL определяет стандартные операции для простых типов UML (кроме UnlimitedNatural) и вводит новый тип Real.

Если модель ориентирована на определенный язык программирования, например Java или C#, можно использовать простые типы этого языка. Однако тогда модель будет привязана к этому конкретному языку программирования.

Иногда необходимо ввести в UML-модель ряд примитивных типов, используемых в остальных классах в качестве атрибутов и типов параметров операций. Добавить простой тип можно, создав класс с таким же именем и стереотипом «primitive». У такого класса обычно нет атрибутов или операций, поскольку он просто выступает в роли структурного нуля для простого типа, добавляя его имя в пространство имен модели.

7.5.2.3. Кратность

Кратность широко используется при проектировании. Она также может быть полезной в аналитических моделях, поскольку с ее помощью можно лаконично обозначить определенные бизнес-ограничения, касающиеся «количества сущностей», участвующих в отношении.

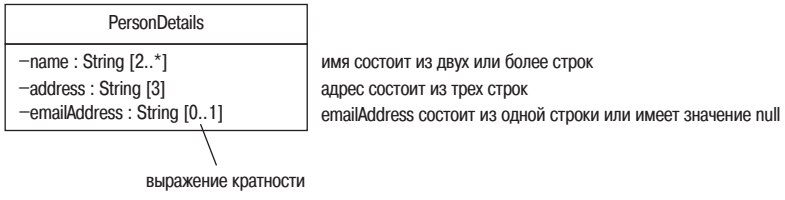


Рис. 7.9. *Примеры синтаксиса кратности*

> Кратность позволяет моделировать коллекции сущностей или неопределенные значения.

Применяя выражение кратности к атрибуту, можно моделировать две совершенно разные сущности (рис. 7.9).

- Коллекции – если выражение кратности определяет целое число больше нуля, значит, задается коллекция типа. Например, выражение colors : Color[7] моделирует атрибут, являющийся коллекцией из семи объектов Color (цвет), которые можно использовать для моделирования цветов радуги.
- Неопределенные значения – большинство языков различают атрибут, содержащий пустой, или неинициализированный, объект (например, пустая строка, ""), и атрибут, указывающий в «никуда», т. е. ссылку на неопределенный (null) объект. Когда атрибут ссылается на null, это означает, что объект, на который он указывает, еще не был создан или уже прекратил существование. При детальном проектировании иногда важно указать, что null является возможным значением атрибута. Смоделировать это можно с помощью специального выражения кратности [0..1]. Рассмотрим пример атрибута emailAddress (электронныйАдрес) (рис. 7.9): если значение атрибута – "" (пустая строка), то это можно понять так, что адрес электронной почты был запрошен, но не был предоставлен. С другой стороны, если атрибут emailAddress указывает на null, можно сказать, что адрес электронной почты еще не был запрошен и, следовательно, его значение неизвестно. Если проводятся такие разграничения, *исключительно важно* указывать в документации emailAddress точную семантику "" и null. Как видите, это довольно тонкие проектные рассуждения, но они могут быть важны и полезны.

На рис. 7.9 приведено несколько примеров синтаксиса кратности.

7.5.2.4. Начальное значение

> Начальное значение позволяет задавать значение атрибута в момент создания объекта.

Начальное значение (initialValue) позволяет задавать значение, принимаемое атрибутом при создании экземпляра класса (т. е. объекта). Это значение называют начальным, потому что оно присваивается атрибуту в момент создания объекта. Использование начальных значений везде, где есть возможность, является хорошим стилем программирования. Это гарантирует, что состояние создаваемых объектов класса всегда действительное (valid) и они пригодны к использованию.

В анализе начальные значения используются только тогда, когда они могут выразить или обозначить важное бизнес-ограничение. Подобная ситуация встречается довольно редко.

7.5.2.5. Расширенный синтаксис атрибута

Подобно любому другому элементу моделирования UML, семантику атрибутов можно расширять, указывая перед ними стереотипы или задавая после них помеченные значения, например

```
«стереотип» атрибут { метка1 = значение1, метка2 = значение2, ... }
```

В помеченных значениях можно хранить любую информацию. Например, иногда они используются для хранения информации о версии, как показано здесь:

```
address { addedBy="Jim Arlow", dateAdded="20MAR2004" }
```

В этом примере записано, что Джим Арлоу (Jim Arlow) добавил атрибут address (адрес) в некоторый класс 20 марта 2004 года.

7.5.3. Ячейка операций

Операции – это функции, закрепленные за определенным классом. По сути, они обладают всеми характеристиками функций:

- имя
- список параметров
- возвращаемый тип

> Сигнатура операции включает имя, тип всех ее параметров и возвращаемый тип.

Сочетание имени операции, типов всех ее параметров и возвращаемого типа образует сигнатуру операции (рис. 7.10). Сигнатура каждой опе-

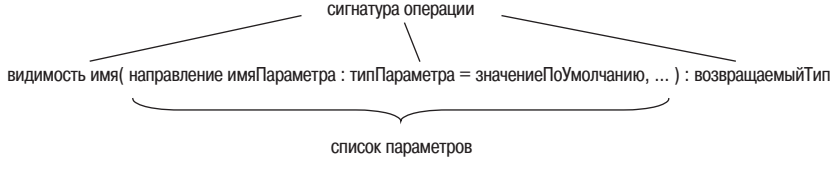


Рис. 7.10. *Сигнатура операции*

рации класса должна быть уникальной, поскольку именно сигнатура идентифицирует операцию. Когда объект получает сообщение, его сигнатура сравнивается с сигнатурами операций, определенных в классе объекта. Если обнаруживается соответствие, инициируется запрашиваемая операция объекта.

Разные языки программирования немного по-разному определяют, что должно составлять сигнатуру операции. Например, в C++ и Java возвращаемый тип игнорируется. Это означает, что две операции класса, отличающиеся только возвращаемыми типами, будут рассматриваться как одна и та же операция и генерировать ошибку компилятора/интерпретатора. В Smalltalk, языке со слабым контролем типов, все параметры и возвращаемый тип имеют тип Object, и поэтому сигнатуру составляет только имя операции.

Имена операций записываются в стиле lowerCamelCase. В качестве имен используются глаголы или глагольные группы. Необходимо избегать применения в именах специальных символом и сокращений.

Каждый параметр операции имеет форму, показанную на рис. 7.11. В операции может быть от нуля до нескольких параметров.

направление имяПараметра : типПараметра = значениеПоУмолчанию

Рис. 7.11. *Синтаксис параметров операции*

Имена параметров записываются в стиле lowerCamelCase и обычно являются именем существительным или именной группой, поскольку обозначают что-то, что передается в операцию. Каждый параметр имеет определенный тип, который является классом или примитивным типом.

Направление и применяемое по умолчанию значение параметра рассматриваются в следующих двух разделах.

7.5.3.1. Направление параметра

Для параметров операции может быть задано направление:

```
операция( in p1:Integer, inout p2:Integer, out p3:Integer, return p4:Integer, return p5:Integer )
```

Если направление не указано, по умолчанию оно принимает значение in (в). Семантика направления параметра приведена в табл. 7.6.

Семантика значений in, inout (в-из) и out (из) довольно проста, а вот return (возвратить) возможно потребует большего внимания.

Обычно ожидается, что функция возвращает один объект, как показано ниже:

```
maximumValue = max( a, b )
minimumValue = min( a, b )
```

где a, b, maximumValue (максимальное значение) и minimumValue (минимальное значение) – целые числа. Обычный синтаксис операции поддерживает именно эту ситуацию: каждая операция обычно имеет только одно возвращаемое значение. Как мы видели, в UML такие «обычные» операции можно смоделировать следующим образом:

```
maximumValue( a:Integer, b:Integer ) : Integer
```

Таблица 7.6

Параметр	Семантика
in p1:Integer	Применяется по умолчанию. Операция использует p1 как входной параметр. Значение p1 каким-то образом используется операцией. Операция *не* изменяет p1.
inout p2:Integer	Операция принимает p2 как параметр ввода/вывода. Значение p2 каким-то образом используется операцией. *и* принимает выходное значение операции. Операция может изменять p2.
out p3:Integer	Операция использует p3 как выходной параметр. Параметр служит хранилищем для выходного значения операции. Операция может изменять p3.
return p4:Integer	Операция использует p4 как возвращаемый параметр. Операция возвращает p4 как одно из своих возвращаемых значений.
return p5:Integer	Операция использует p5 как возвращаемый параметр. Операция возвращает p5 как одно из своих возвращаемых значений.

Однако в некоторых языках операции могут возвращать *более* одного значения. Например, в языке программирования Python можно было бы написать:

```
maximumValue, minimumValue = maxMin( a, b )
```

где maxMin(a, b) возвращает два значения – максимальное и минимальное. В UML это можно смоделировать следующим образом:

```
maxMin( in a: Integer, in b:Integer, return maxValue:Integer, return minValue:Integer )
```

Как видим, направление return параметра позволяет моделировать ситуации, когда операция возвращает *более одного* значения.

Возвращаемые параметры могут быть перечислены и после имени операции:

```
maxMin( in a: Integer, in b:Integer ) : Integer, Integer
```

В данном случае применение ключевого слова return было бы плохим стилем, поскольку возвращаемые параметры и так явно выделены.

На самом деле направления параметров являются вопросом проектирования, поэтому обычно во время анализа о них не заботятся (если только не используется OCL).

Направления параметров имеют большое значение при проектировании, особенно если речь идет о генерации кода. Генераторы кода будут проецировать направления параметров UML в конкретную семантику передачи параметров в целевом языке программирования.

7.5.3.2. Значения параметров по умолчанию

Для параметра операции может быть задано значение, применяемое по умолчанию. Если при вызове операции параметру не присваивается значение, используется значение по умолчанию.

На рис. 7.12 показано, как задаются применяемые по умолчанию значения. Здесь в классе Canvas имеется две операции: drawCircle(...) (нарисовать круг) и drawSquare(...) (нарисовать квадрат), отрисовывающие на экране круг и квадрат соответственно. Для параметра origin (начало отсчета) задается значение по умолчанию Point(0,0). Если при вызове этих операций данный параметр будет опущен, фигуры будут отрисованы на экране относительно точки {0,0}.

На самом деле применяемые по умолчанию значения являются вопросом проектирования; во время анализа они используются редко.

7.5.3.3. Расширенный синтаксис операции

Семантику операций можно расширить, предваряя их стереотипами и дополняя помеченными значениями.

«стереотип» операция(...) { метка1 = значение1, метка2 = значение2, ... }

Можно также ввести помеченные значения в параметры операций, но мы ни разу не сталкивались с ситуацией, когда это было бы полезно.

7.5.3.4. Операции запроса

В каждой операции есть свойство isQuery. Если в инструменте моделирования этому свойству присвоено значение true, операция является операцией запроса. Это означает, что она *не имеет побочных эффектов* и не меняет состояние объекта, в котором вызывается. Операция, воз-

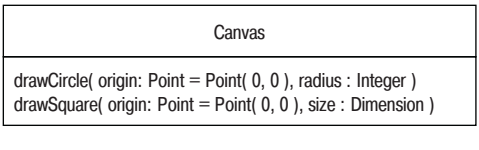


Рис. 7.12. *Задание параметров по умолчанию*

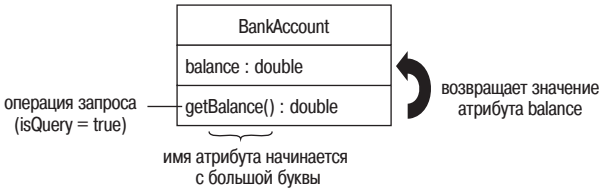


Рис. 7.13. *Именование операций запроса*

вращающая значение атрибута, называется операцией запроса. Ее свойству isQuery должно быть присвоено значение true.

По умолчанию isQuery имеет значение false. Обычно isQuery задается при проектировании. Однако если в UML-моделях применяется OCL (см. главу 25), важно обозначить операции запроса, потому что выражения OCL не могут менять состояния системы и поэтому могут использовать *только* операции запроса. Если задано isQuery, компиляторы OCL могут проверять, какие операции (допустимые или нет) вызывают OCL-выражения.

Более или менее универсальный способ наименования операций запроса – ставить приставку get перед именем того, что вы запрашиваете. Пример приведен на рис. 7.13.

7.5.4. Синтаксис стереотипа класса

Стереотипы могут отображаться по-разному (рис. 7.14). Однако большинство разработчиков моделей используют имя, заключенное в кавычки («имя стереотипа»), или пиктограмму. Другие варианты не так широко используются, и инструментальные средства моделирования часто налагают ограничения.

Стереотипы также могут быть ассоциированы с цветами или текстурами, но это очень плохая практика. У некоторых читателей, имеющих проблемы со зрением или не различающих цвета, могут возникнуть сложности с пониманием таких диаграмм. Кроме того, чаще всего диаграммы распечатываются в черно-белом варианте.

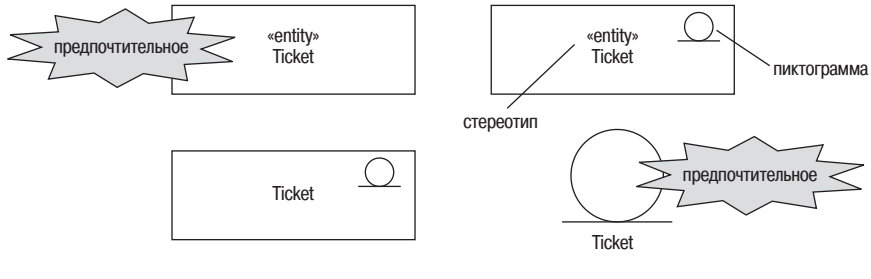


Рис. 7.14. *Варианты отображения стереотипов*

7.6. Область действия

> Атрибуты и операции, область действия которых – экземпляр, принадлежат или выполняются определенным объектом.

До сих пор мы видели, что у объектов есть собственные копии атрибутов, определенных в их классе. Таким образом, атрибуты разных объектов могут принимать разные значения. Аналогично все операции, рассмотренные до сих пор, выполняются определенными объектами. Это обычная ситуация. Говорят, что эти атрибуты и операции имеют *область действия – экземпляр*.

> Атрибуты и операции, область действия которых – класс, принадлежат или выполняются во всех объектах данного класса.

Однако иногда нужно определить атрибуты, которые имеют единственное, общее для *всех* объектов класса значение. И нужны операции (как операции создания объектов), не относящиеся ни к одному конкретному экземпляру класса. Говорят, что такие атрибуты и операции имеют *область действия – класс*. Свойства, область действия которых – класс, обеспечивают набор глобальных характеристик объектов класса.

7.6.1. Область действия – класс и область действия – экземпляр

Обозначение атрибутов и операций с областями действия класс и экземпляр показано на рис. 7.15. Семантика таких атрибутов и операций приведена в табл. 7.7.

7.6.2. Область действия определяет возможность доступа

Возможность доступа операции к другому свойству класса определяется областью действия операции и областью действия свойства, к которому пытаются получить доступ.

Операции, область действия которых – экземпляр, могут организовывать доступ к другим атрибутам и операциям с такой же областью дей-

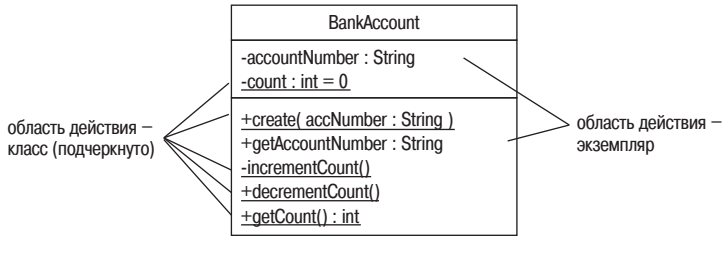


***Рис. 7.15.** Атрибуты и операции с областями действия класс и экземпляр*

ствия, а также ко всем атрибутам и операциям, область действия которых – класс.

Операции, область действия которых – класс, могут организовывать доступ *только* к атрибутам и операциям, имеющим область действия – класс. Операции уровня класса не имеют доступа к операциям уровня экземпляра, потому что:

- возможно, еще не создано ни одного экземпляра класса;
- даже если экземпляры класса существуют, неизвестно, какой из них использовать.

Таблица 7.7

	Область действия – экземпляр	**Область действия – класс**
Атрибуты	По умолчанию область действия атрибутов – экземпляр.	Для атрибутов может быть определена область действия класс.
	Каждый объект класса получает собственную копию атрибутов, область действия которых – экземпляр.	Каждый объект класса использует одну и ту же единственную копию атрибутов класса.
	Следовательно, значения атрибутов экземпляра могут быть разными в каждом объекте.	Следовательно, значения атрибутов класса во всех объектах одинаковые.
Операции	По умолчанию область действия операций – экземпляр.	Для операций может быть определена область действия – класс.
	Каждый вызов операции экземпляра касается конкретного экземпляра класса.	Вызов операции класса не касается конкретного экземпляра класса. Операции уровня класса можно рассматривать как применяемые к самому классу.
	Невозможно вызвать операцию экземпляра, не имея в распоряжении экземпляра класса. Безусловно, это означает, что нельзя использовать операции, область действия которых – экземпляр, для создания объектов этого класса: никогда не получится создать первый объект.	Инициировать операции класса можно, даже не имея экземпляра класса; это идеально подходит для операций создания объектов.

7.7. Создание и уничтожение объектов

> Конструкторы – это специальные операции, создающие новые объекты. Область их действия – класс.

Конструкторы – это специальные операции, создающие новые экземпляры классов. Эти операции *должны* иметь область действия – класс. Если бы они были уровня экземпляра, не было бы возможности создать первый экземпляр класса.

Конструкторы – забота проектирования. Обычно их *не* показывают на аналитических моделях.

В разных языках программирования действуют разные стандарты именования конструкторов. Абсолютно универсальный подход – называть конструктор просто create(…) (создать). Это делает понятным назначение данной операции. Однако языки Java, C# и C++ требуют, чтобы имя конструктора совпадало с именем класса.

У класса может быть несколько конструкторов с одинаковыми именами, но с разным набором параметров. Конструктор без параметров называют *применяемым по умолчанию конструктором* (*default constructor*). Параметры конструктора можно использовать для инициализации значений атрибутов в момент создания объекта.

На рис.7.16 показан простой пример класса BankAccount (банковский счет). При каждом создании объекта BankAccount в качестве параметра в его конструктор должно передаваться значение типа String. Эта строка используется для задания атрибута accountNumber (номер счета) (например, значение "XYZ001002"). Тот факт, что конструктору класса BankAccount необходим параметр, говорит о том, что создать объект BankAccount, *не определив* этот параметр, *нельзя*. Это гарантирует задание атрибута accountNumber каждого объекта BankAccount в момент создания объекта. Это очень хороший стиль.

В аналитических моделях обычно не занимаются конструкторами (и тем более деструкторами). Они не имеют ни влияния, ни отношения к бизнес-семантике класса. Если все-таки есть желание обозначить операции создания, можно ввести операцию create() без параметров как структурный нуль. Или можно указать только параметры, имеющие в перспективе существенное значение.

Когда дело дойдет до детального проекта, необходимо будет определить имя, типы параметров и возвращаемый тип *каждой* операции (см. раздел 17.4). Сюда входит и явное описание конструктора и деструктора.

Уничтожение объекта не такая простая операция, как его создание. В разных ОО языках программирования семантика уничтожения объ-

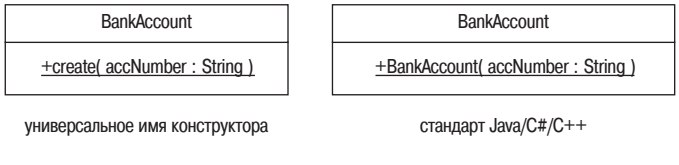


Рис. 7.16. *Именование конструктора*

екта разная. Более подробно создание и уничтожение объектов рассматриваются в следующих двух разделах.

7.7.1. Конструкторы – пример класса ClubMember

Пример класса ClubMember (член клуба) (рис. 7.17) показывает обычное применение атрибутов и операций уровня класса. Этот класс описывает некий член клуба. Атрибут numberOfMembers (количество членов) – закрытый (private) атрибут класса типа int. Следовательно, этот атрибут используется совместно всеми объектами класса ClubMember. Его значение для каждого из этих объектов будет одинаковым.

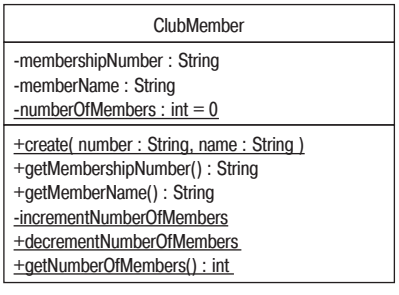


Рис. 7.17. *Применение атрибутов и операций уровня класса*

При создании атрибута numberOfMembers ему присваивается начальное значение, равное нулю. Далее, если бы это был атрибут экземпляра, каждый объект при создании получал бы собственную копию этого атрибута. Однако область действия этого атрибута – класс. Значит, существует только одна его копия, и эта единственная копия инициализируется только один раз. *Когда* именно это происходит, зависит от языка реализации. Мы должны знать лишь то, что атрибут инициализируется со значением нуль при запуске программы.

Предположим, что в операции create(...) происходит вызов операции класса incrementNumberOfmembers() (увеличить число членов). Как можно ожидать из ее имени, эта операция увеличивает значение атрибута класса numberOfMembers. При каждом создании экземпляра класса numberOfMembers увеличивается на единицу. В класс введен счетчик! С помощью операции класса getNumberOfMembers() (получить количество членов) можно запросить значение атрибута numberOfMembers. Данная операция возвращает число, равное количеству созданных объектов ClubMember.

7.7.2. Деструкторы – пример класса ClubMember

> Деструкторы – это специальные операции, которые «наводят порядок» при уничтожении объектов.

Что происходит, если программа создает *и* уничтожает объекты ClubMember? Очевидно, что значение атрибута numberOfMembers быстро потеряет смысл. Исправить эту ситуацию можно, введя в класс операцию для уменьшения значения атрибута numberOfMembers (рис. 7.17) и обеспечив ее вызов при каждом уничтожении экземпляра класса ClubMember.

В некоторых ОО языках программирования есть специальные операции уровня экземпляра, называемые деструкторами, которые автоматически вызываются в момент уничтожения объекта. Например, в C++ деструктор всегда имеет форму ~ИмяКласса (списокПараметров). В C++ операция уничтожения *гарантированно* вызывается в момент уничтожения объекта.

В Java есть аналогичная возможность: каждый класс имеет операцию под названием finalize(), которая вызывается при окончательном уничтожении объекта. Но сама программа явно не уничтожает объекты. Этим занимается автоматический сборщик мусора. Вам известно, что finalize() будет вызван, но вы не знаете, *когда* это произойдет! Конечно, это не подходит для нашего простого приложения со счетчиком. Здесь приходится самостоятельно уменьшать значение атрибута numberOfMembers, вызывая операцию уровня класса decrementNumberOfMembers(), когда работа с объектом завершена и он отправляется в сборщик мусора.

C# имеет аналогичную Java семантику уничтожения, только операция называется Finalize().

7.8. Что мы узнали

В этой главе были представлены основные сведения по классам и объектам, которые используются далее в книге. Классы и объекты – это строительные блоки ОО систем, поэтому важно всесторонне и детально их понимать.

Мы узнали следующее:

- Объекты – это образующие единое целое элементы, сочетающие в себе данные и функции.
- Инкапсуляция – данные, находящиеся внутри объекта, скрыты. Манипулировать ими можно, только инициируя одну из функций объекта.
  - Операции – это спецификации функций объекта, создаваемые во время анализа.
  - Методы – это реализации функций объекта, создаваемые во время реализации.
- Каждый объект – это экземпляр класса. Класс определяет общие характеристики, присущие всем объектам этого класса.

- У каждого объекта есть следующие характеристики:
  - Идентичность – уникальность существующего объекта: вы используете объектные ссылки для однозначного указания на конкретный объект.
  - Состояние – значимый набор значений атрибутов и отношений объекта в определенный момент времени.
    - Состояние образуют наборы только таких значений атрибутов и отношений, которые обуславливают существенное семантическое отличие от других возможных наборов. Например, объект BankAccount: balance < 0, state = Overdrawn; balance > 0, state = InCredit.
    - Переход состояний – перемещение объекта из одного значимого состояния в другое.
  - Поведение – сервисы, предлагаемые объектом другим объектам:
    - моделируется в виде набора операций;
    - вызов операции *может* генерировать переход состояния.
- Совместная работа объектов генерирует поведение системы.
  - Взаимодействие включает в себя обмен сообщениями между объектами – при получении сообщения инициируется соответствующая операция; это *может* привести к переходу состояний.
- Нотация объектов в UML – в каждой пиктограмме объекта две ячейки.
  - В верхней ячейке находится имя объекта и/или имя класса, *и то и другое* должно быть подчеркнуто.
    - Имена объектов записываются в стиле lowerCamelCase.
    - Имена классов записываются в стиле UpperCamelCase.
    - Никаких специальных символов, знаков препинания или сокращений.
    - Имя объекта отделяется от имени класса одним двоеточием.
  - Нижняя ячейка содержит имена и значения атрибутов, разделяемые знаком равенства.
    - Имена атрибутов записываются в стиле lowerCamelCase.
    - Типы атрибутов могут быть представлены, но обычно для краткости их опускают.
- Класс определяет характеристики (атрибуты, операции, отношения и поведение) ряда объектов.
  - Каждый объект – это экземпляр только одного класса.
  - Разные объекты одного класса имеют одинаковый набор атрибутов, но значения этих атрибутов могут быть разными. Разные значения атрибутов обусловливают разное поведение объектов одного класса. Например, сравните попытку снять $100 с объек-

та BankAccount, кредит которого превышен, с попыткой снять $100 с объекта BankAccount, на котором есть $200 кредита.

- Существует множество классификаций мира. Правильный выбор схемы классификации – один из ключей к успешному ОО анализу.
- Отношение создания экземпляра между классом и одним из его объектов можно показать с помощью зависимости, обозначенной стереотипом «instantiate»:
  - отношения объединяют сущности;
  - отношение зависимости показывает, что изменение сущности-поставщика оказывает влияние на сущность-клиент.
- При создании экземпляра с использованием класса в качестве шаблона создается новый объект.
  - Большинство ОО языков программирования предоставляют специальные операции, конструкторы, которые вызываются, когда необходимо создать объект – конструкторы создают или инициализируют объекты; область действия конструкторов – класс (они принадлежат классу).
  - Некоторые ОО языки программирования предоставляют специальные операции, деструкторы, которые вызываются при уничтожении объекта – деструкторы «наводят порядок» после уничтожения объектов.

- Нотация классов в UML:
  - В ячейке имени размещается имя класса, записанное в стиле UpperCamelCase:
    - никаких сокращений, знаков препинания или специальных символов.
  - Ячейка атрибутов – у каждого атрибута есть:
    - видимость – управляет доступом к характеристикам класса;
    - имя (обязательно) – записывается в стиле lowerCamelCase;
    - кратность – коллекции, например [10]; неопределенные значения, например [0..1];
    - тип;
    - у атрибутов могут быть стереотипы и помеченные значения.
  - Ячейка операций – у каждой операции может быть:
    - видимость;
    - имя (обязательно) – записывается в стиле lowerCamelCase;
    - список параметров (имя и тип каждого параметра);
      - у параметра может быть применяемое по умолчанию значение (не обязательно);

        - у параметра может быть направление (не обязательно): in, out, inout, return.
      - возвращаемый тип;
      - стереотип;
      - помеченные значения.
    - В операциях запроса атрибут isQuery = true – эти операции не оказывают побочного действия.
    - Сигнатура операции включает:
      - имя;
      - список параметров (типы всех параметров);
      - возвращаемый тип.
    - Сигнатура каждой операции или метода класса должна быть уникальной.
- Область действия.
  - Атрибуты и операции, область действия которых – экземпляр, принадлежат или выполняются в конкретных объектах:
    - операции уровня экземпляра имеют доступ к другим операциям или атрибутам уровня экземпляра;
    - операции уровня экземпляра имеют доступ ко всем атрибутам или операциям уровня класса.
  - Атрибуты и операции, область действия которых – класс, принадлежат или выполняются во всех объектах класса:
    - атрибуты и операции уровня класса имеют доступ только к операциям уровня класса.

8

Выявление классов анализа

8.1. План главы

Эта глава посвящена основной деятельности ОО анализа – выявлению классов анализа. В разделе 8.2 описывается деятельность UP по выявлению классов анализа. Из раздела 8.3 можно узнать, что такое класс анализа.

В разделе 8.4 обсуждается, как выявлять классы анализа. Представлены три метода: анализ существительное/глагол (8.4.1), анализ CRC (8.4.2) и RUP-стереотипы классов анализа (8.4.3). В подразделе 8.4.4 рассмотрены в общих чертах другие возможные источники классов.

8.2. Деятельность UP: Анализ прецедента

Результатом рабочего потока UP Анализ прецедента (рис. 8.2) являются классы анализа и реализации прецедентов. В этой главе основное внимание сосредоточено на классах анализа. Реализации прецедентов (глава 12) – это кооперации объектов, показывающие, как системы взаимодействующих объектов могут реализовывать поведение системы, описанное в прецеденте.

Рассмотрим входные данные Анализа прецедента.

> Деятельность UP «Анализ прецедента» включает создание классов анализа и реализации прецедентов.

- Бизнес-модель – в распоряжении разработчиков модели может быть (а может и не быть) бизнес-модель моделируемой системы. Если она уже есть, это превосходный источник требований.
- Модель требований – процесс создания этой модели описан в главе 3. Требования (этот артефакт затушеван, чтобы показать изменение по сравнению с исходным рисунком) обеспечивают полезные вход-

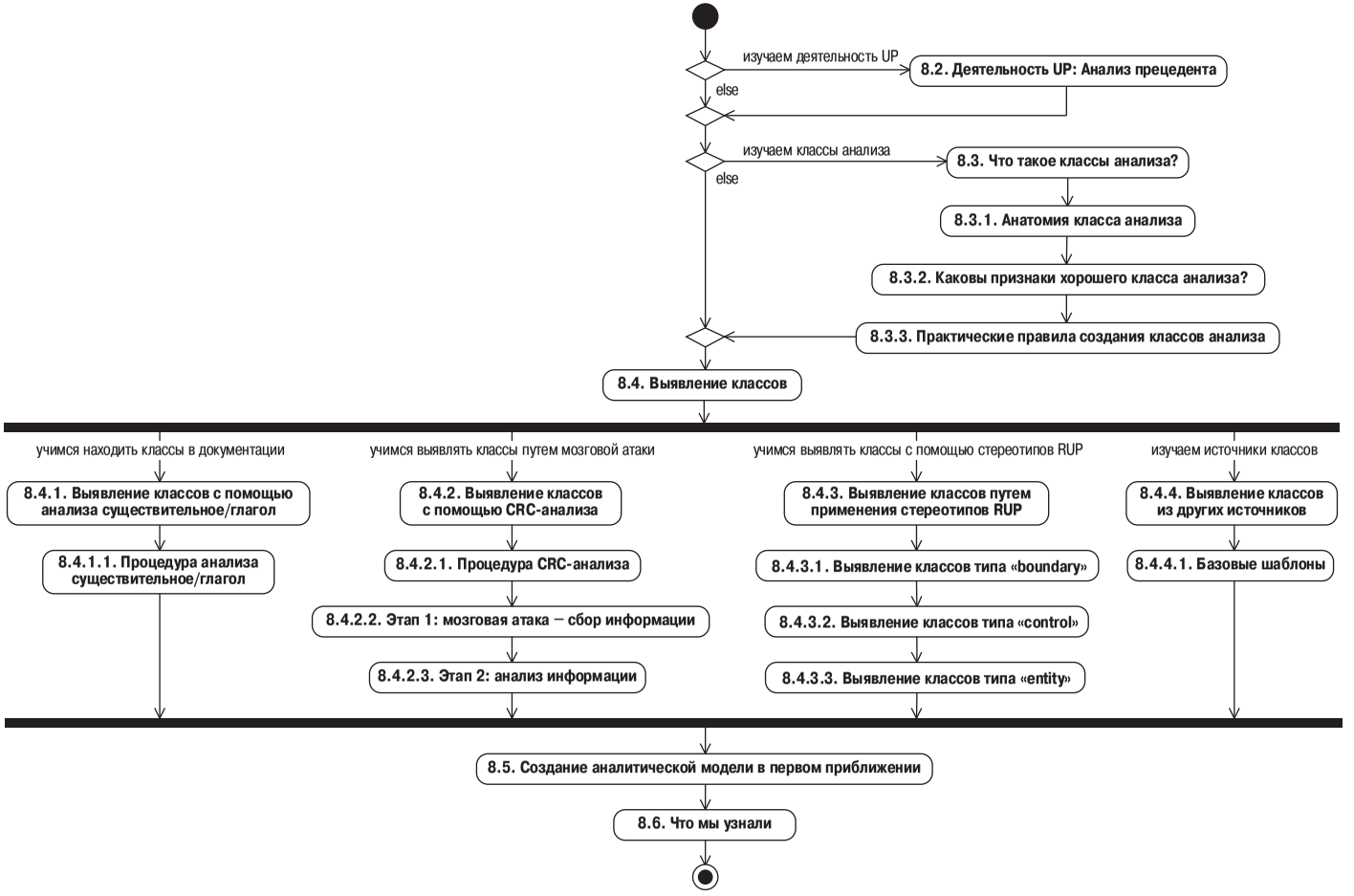


Рис. 8.1. *План главы*

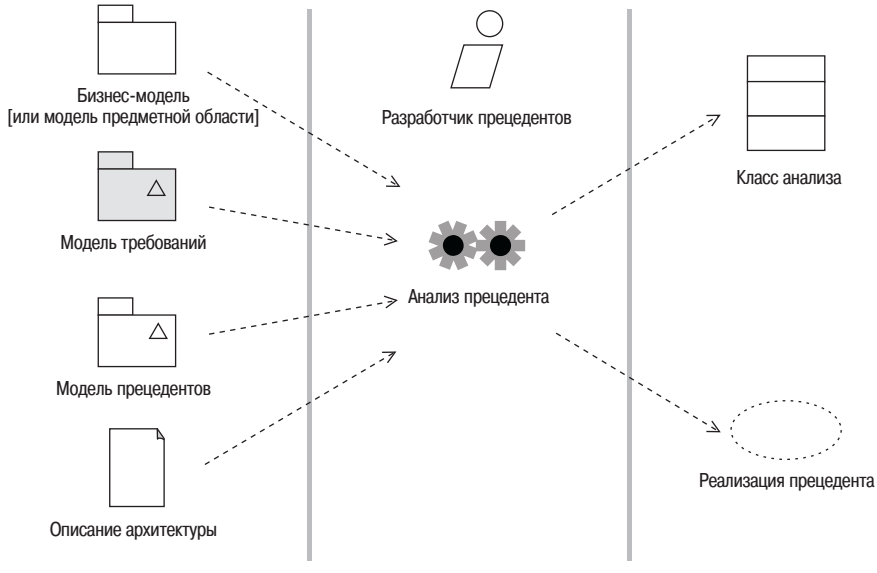


Рис. 8.2. *Деятельность UP: Анализ прецедента. Адаптировано с рис. 8.25 [Jacobson 1] с разрешения издательства Addison-Wesley*

ные данные для процесса моделирования прецедентов. В частности, функциональные требования предложат прецеденты и актеров, а нефункциональные требования – то, что, возможно, надо иметь в виду при создании модели прецедентов.

- Модель прецедентов – создание модели прецедентов обсуждалось в главах 4 и 5.
- Описание архитектуры – представление важных с архитектурной точки зрения аспектов системы. Может включать фрагменты UML-моделей, вставленные в пояснительный текст. Создается архитекторами на основании данных, полученных от аналитиков или проектировщиков.

8.3. Что такое классы анализа?

> Классы анализа моделируют важные аспекты предметной области, такие как «покупатель» или «продукт».

Классы анализа – это классы, которые:

- представляют четкую абстракцию предметной области (problem domain);
- должны проецироваться на реальные бизнес-понятия (и быть аккуратно поименованы соответственно этим понятиям).

Предметная область – это область, в которой возникает необходимость в программной системе (и, следовательно, в деятельности по разработке программного обеспечения). Обычно это определенная область деловой активности, например сетевая торговля или управление взаимоотношениями с клиентами. Однако важно отметить, что предметная область может вообще не быть деловой активностью, а являться следствием существования физического оборудования, для которого необходимо программное обеспечение. В конечном счете, вся разработка коммерческого программного обеспечения служит некоторой прикладной цели, будь то автоматизация существующего бизнес-процесса или разработка нового продукта, имеющего существенную программную составляющую.

> Класс анализа должен четко и однозначно проецироваться в реальное прикладное понятие.

Самое важное свойство класса анализа – он должен четко и однозначно проецироваться в некоторое реальное прикладное понятие, например покупатель, продукт или счет. Это предполагает четкость и однозначность самих бизнес-понятий, что является большой редкостью. Следовательно, задача ОО аналитика – попытаться прояснить беспорядочные или несоответствующие прикладные понятия и превратить их в то, что может стать основой для класса анализа. Вот в чем сложность ОО анализа.

Итак, первый шаг в построении ОО программного обеспечения – прояснить предметную область. Если она содержит четко определенные бизнес-понятия и имеет простую функциональную структуру, решение практически лежит на поверхности. Большая часть этой работы осуществляется в рабочем потоке определения требований в деятельностях по выявлению требований и созданию модели прецедентов и глоссария проекта. Однако намного большая ясность вносится при построении классов анализа и реализации прецедентов.

Важно, чтобы *все* классы аналитической модели являлись классами анализа, а не классами, вытекающими из проектных соображений (области решения). Когда дело дойдет до детального проекта, может случиться так, что классы анализа будут в конце концов переработаны в один или более проектных классов.

> Правильное выявление классов анализа – ключ к ОО анализу и проектированию.

В предыдущей главе мы поневоле начинали с рассмотрения конкретных объектов. Теперь вы поймете, что подлинная цель ОО анализа – выявление *классов* этих объектов. По сути, правильное выявление классов анализа – ключ к ОО анализу и проектированию. Если при анализе классы определены неверно, то и весь процесс производства программ-

ного обеспечения, основывающийся на рабочих потоках определения требований и анализа, окажется под угрозой. Поэтому критически важно уделить анализу достаточное количество времени, чтобы быть уверенными в правильности определения классов анализа. И это время будет потрачено не зря, поскольку практически наверняка оно сэкономит время в дальнейшем.

В этой книге основное внимание уделяется разработке бизнес-систем, поскольку именно этим занимается большинство ОО аналитиков и проектировщиков. Но разработка встроенных систем – лишь особый случай разработки обычной бизнес-системы, поэтому к ней применимы те же основные принципы. В бизнес-системах обычно доминируют функциональные требования, поэтому самым сложным здесь является определение требований и анализ. Во встроенных системах, как правило, доминируют нефункциональные требования, вытекающие из аппаратных средств, в которые встраивается система. В этом случае анализ довольно прост, тогда как проектирование может быть довольно сложным. Требования важны для *всех* типов систем, а для некоторых встроенных систем, таких как устройства управления рентгеновскими аппаратами, они могут стать вопросом жизни и смерти.

8.3.1. Анатомия класса анализа

Классы анализа должны представлять «высокоуровневый» набор атрибутов. Они у*казывают* атрибуты, которые, *возможно*, будут присутствовать в проектных классах. Можно сказать, что классы анализа включают предполагаемые атрибуты проектных классов.

> В классах анализа содержатся только ключевые атрибуты и обязанности, определенные на очень высоком уровне абстракции.

Операции классов анализа определяют на высоком уровне абстракции, ключевые сервисы, которые должен предлагать класс. В проекте они станут реальными операциями. Однако одна операция уровня анализа очень часто разбивается на несколько операций уровня проекта.

Синтаксис UML уже подробно обсуждался в главе 7. В анализе реально используется лишь небольшая его часть. Конечно, аналитик всегда волен добавить любые необходимые, по его мнению, дополнения, чтобы сделать модель более понятной. Однако базовый синтаксис класса анализа *всегда* избегает деталей реализации. В конце концов, при анализе создается общее представление.

Минимальная форма класса анализа включает следующее?

- Имя – обязательно.
- Атрибуты – имена атрибутов являются обязательными, хотя на данном этапе могут моделироваться только важные предполагаемые атрибуты. Типы атрибутов считаются необязательными.

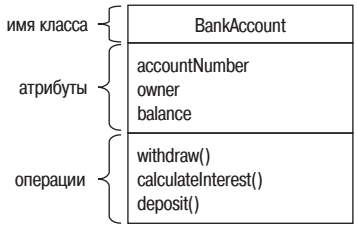


Рис. 8.3. *Пример класса анализа*

- Операции – в анализе операции могут быть всего лишь очень приблизительными формулировками обязанностей класса. Параметры и возвращаемые типы операций приводятся только в том случае, если они важны для понимания модели.
- Видимость – обычно не указывается.
- Стереотипы – могут указываться, если это приводит к улучшению модели.
- Помеченные значения – могут указываться, если это улучшает модель.

Пример класса анализа приведен на рис. 8.3.

Основное назначение класса анализа состоит в том, что в нем делается попытка уловить *суть* абстракции, а детали реализации оставляют на этап проектирования.

8.3.2. Каковы признаки хорошего класса анализа?

Признаки хорошего класса анализа можно кратко охарактеризовать следующим образом:

- его имя отражает его назначение;
- он является четкой абстракцией, моделирующей один конкретный элемент предметной области;
- он проецируется на четко определяемую возможность предметной области;
- у него небольшой четко определенный набор обязанностей;
- у него высокая внутренняя связность (cohesion);
- у него низкая связанность с другими классами (coupling).

> Имя класса анализа должно отражать его назначение.

При анализе делается попытка точно и лаконично смоделировать один аспект предметной области с точки зрения создаваемой системы. Например, при моделировании клиента банковской системы необходимо записать его имя, адрес и т. д. При этом вряд ли представляет интерес

информация о том, предпочитает ли он сидеть в самолете у окна или в проходе. Необходимо сосредоточиться на важных с точки зрения строящейся системы аспектах реальных сущностей.

Нередко составить первое впечатление о том, «хорош» класс или нет, можно уже по его имени. В системе электронной коммерции вполне понятно, какую реальную сущность может обозначать имя Customer; оно является хорошей кандидатурой для имени класса. ShoppingBasket также могло бы быть хорошей абстракцией, мы практически интуитивно понимаем его семантику. Однако семантика такого имени, как WebSiteVisitor (посетитель веб-сайта), кажется неопределенной и по сути напоминает роль, выполняемую Customer по отношению к системе электронной коммерции. Необходимо всегда искать «четкую абстракцию» – что-то, что имеет ясную и очевидную семантику.

> Обязанности описывают связный набор операций.

Обязанность – это контракт или обязательство класса по отношению к его клиентам. По существу, обязанность – это сервис, который класс предлагает другим классам. Очень важно, чтобы классы анализа имели связный набор обязанностей, абсолютно соответствующих назначению класса (которое выражено в его имени) и реальной «сущности», моделируемой классом. Возвращаясь к примеру ShoppingBasket, можно ожидать, что этот класс будет иметь следующие обязанности:

- добавить позицию в корзину;
- удалить позицию из корзины;
- показать позиции, находящиеся в корзине.

Это связный набор обязанностей, обслуживающий коллекцию товарных позиций, выбранных покупателем. Он является связным, потому что все обязанности направлены на реализацию одной цели – обслуживание корзины для покупок клиента. На очень высоком уровне абстракции эти три обязанности можно представить как обязанность «обслужить корзину».

Теперь в класс ShoppingBasket можно было бы добавить следующие обязанности:

- проверить действительность кредитной карты;
- принять платеж;
- распечатать квитанцию.

Но эти обязанности, похоже, не соответствуют назначению или интуитивно понятной семантике корзин для покупок. Они не являются связными и, без сомнения, должны быть заданы где-нибудь в другом месте – может быть, в классах CreditCardCompany (компания, продающая товары по кредитным карточкам), Checkout (касса) и ReceiptPrinter (устройство для распечатки квитанций). Важно правильно распределить

обязанности между классами анализа, чтобы обеспечить максимальную внутреннюю связность каждого класса.

И наконец, хорошие классы обладают минимальной связанностью (coupling) с другими классами. Связанность измеряется количеством классов, с которыми данный класс имеет отношения. Равномерное распределение обязанностей между классами обеспечит в результате низкую связанность. Размещение управления или множества обязанностей в одном классе приводит к повышению связанности этого класса. Способы максимального повышения внутренней связности и уменьшения связанности рассматриваются в главе 15.

8.3.3. Практические правила создания классов анализа

Здесь приведены некоторые практические правила создания правильно сформированных классов анализа.

- В каждом классе должно быть три-пять обязанностей – необходимо стремиться сохранить максимальную простоту классов. Обычно число обязанностей, которые они могут поддерживать, ограничивают тремя-пятью. Приводимый ранее класс ShoppingBasket является хорошим примером класса с небольшим и управляемым количеством обязанностей.
- Ни один класс не является изолированным – суть хорошего ОО анализа и проектирования во взаимодействии классов друг с другом с целью предоставить пользователям значимый результат. По существу, каждый класс должен быть ассоциирован с небольшим количеством других классов, взаимодействуя с которыми он обеспечивает требуемый результат. Классы могут делегировать некоторые из своих обязанностей другим «вспомогательным» классам, предназначенным для выполнения именно этой конкретной функции.
- Остерегайтесь создания множества очень мелких классов – иногда это может нарушить баланс распределения обязанностей. Если в модели масса мелких классов, каждый из которых имеет одну-две обязанности, необходимо тщательно проанализировать ситуацию с целью объединения нескольких мелких классов в большие.
- Следует опасаться и противоположной ситуации, когда в модели несколько очень больших классов, многие из которых обладают большим числом (> 5) обязанностей. Стратегия в этом случае такова: по очереди рассмотреть эти классы и проанализировать возможность их разбиения на несколько меньших классов с допустимым количеством обязанностей.
- Остерегайтесь «функтоидов»; функтоид – это на самом деле обычная процедурная функция, выдаваемая за класс. Гради Буч любит рассказывать забавный анекдот о модели очень простой системы, состоящей из тысяч классов. При тщательном рассмотрении оказалось, что в каждом классе была всего одна операция doIt() (сделай

«это»). Опасность функтоидов всегда существует, когда аналитики, привыкшие к прямой функциональной декомпозиции, впервые берутся за ОО анализ и проектирование.

- Опасайтесь всемогущих классов – классов, которые, кажется, делают все. Ищите классы, в именах которых есть слова «system» (система) или «controller» (контроллер)! Для решения этой проблемы необходимо посмотреть, можно ли разбить обязанности всемогущего класса на связные подмножества. Если да, то, вероятно, каждый из этих связных наборов обязанностей можно выделить в отдельный класс. Тогда поведение, предлагаемое исходным всемогущим классом, можно было бы получить за счет взаимодействия этих меньших классов.
- Необходимо избегать «глубоких» деревьев наследования – суть проектирования хорошей иерархии наследования в том, что каждый уровень абстракции должен иметь четко определенное назначение. Очень легко создать много по сути бесполезных уровней. Широко распространенной ошибкой является использование иерархии для реализации некоторого рода функциональной декомпозиции, где каждый уровень абстракции имеет только одну обязанность. Это нецелесообразно во всех отношениях, поскольку ведет к запутанной, сложной для понимания модели. В анализе наследование используется только там, где есть явная и четкая иерархия наследования, проистекающая непосредственно из предметной области.

Что касается последнего правила, то следует пояснить, что имеется в виду под понятием «глубокое» дерево наследования. В анализе, где классы представляют бизнес-сущности, «глубокое» – это три и более уровней наследования. Это объясняется тем, что бизнес-сущности имеют тенденцию формировать «широкие», а не «глубокие» иерархии наследования.

При проектировании, когда дерево составляют классы области решения, определение «глубины» зависит от предполагаемого языка реализации. В Java, C++, C#, Python и Visual Basic под «глубоким» понимают дерево наследования, содержащее три или более уровней. Однако в Smalltalk деревья наследования могут быть намного более глубокими из-за структуры системы Smalltalk.

8.4. Выявление классов

Далее в этой главе обсуждается основной вопрос ОО анализа и проектирования – выявление классов анализа.

Как указывает Мейер (Meyer) в книге «Object Oriented Software Construction» [Meyer 1], нет простого алгоритма правильного выявления классов анализа. Наличие подобного алгоритма было бы равнозначно существованию универсального способа разработки ОО программного

обеспечения, что так же невероятно, как появление универсального способа доказательства математических теорем.

Тем не менее есть проверенные и протестированные методы, обеспечивающие хороший результат. Они представлены ниже и включают анализ текста и интервьюирование пользователей и специалистов предметной области. Но, разумеется, несмотря на все эти методы, правильное выявление классов зависит от подхода, мастерства и опыта конкретного аналитика.

8.4.1. Выявление классов с помощью анализа существительное/глагол

Анализ существительное/глагол – очень простой способ анализа текста с целью выявления классов, атрибутов и обязанностей. По сути, существительные и именные группы, встречающиеся в тексте, указывают на обязанности или атрибуты класса, а глаголы и глагольные группы указывают на ответственности и операции класса. Анализ существительное/глагол успешно применяется в течение многих лет, поскольку основывается на прямом анализе языка предметной области. Однако необходимо помнить о синонимах и омонимах, поскольку они могут стать причиной появления ложных классов.

> В анализе существительное/глагол анализируется текст. Существительные и именные группы указывают на классы или атрибуты. Глаголы и глагольные группы служат признаком обязанностей или операций.

Также надо быть очень осторожными, когда предметная область недостаточно понятна и определена. В этом случае необходимо попытаться собрать максимум информации об этой области у максимально возможного числа людей, изучить аналогичные предметные области за пределами вашей организации.

Наверное, самый сложный аспект анализа существительное/глагол – выявление «скрытых» классов. Это классы, которые свойственны предметной области, но могут никогда не упоминаться явно. Например, в системе резервирования для компании, занимающейся организацией отдыха, заинтересованные стороны будут говорить о резервировании, бронировании и т. д., но более важная абстракция, Order (Заказ), может вообще не упоминаться открыто, если ее нет в существующих бизнес-системах. Обычно признаком выявления скрытого класса является «загустевание» системы после введения единственной новой абстракции. Это происходит на удивление часто; на практике, если у нас возникают хоть какие-нибудь трудности с аналитической моделью, мы продолжаем поиск скрытых файлов. Даже если ничего не будет выявлено, это заостряет внимание на некоторых серьезных вопросах и улучшает понимание предметной области.

8.4.1.1. Процедура анализа существительное/глагол

Первый шаг в анализе существительное/глагол – собрать максимально возможный объем относящейся к делу информации. Хорошими источниками информации являются:

- модель требований;
- модель прецедентов;
- глоссарий проекта;
- что-либо еще (архитектура, документы-концепции и т. д.).

После сбора документации проводится ее анализ. Делается это очень просто: выделяется (или регистрируется каким-либо иным способом) следующее:

- существительные, например рейс;
- именные группы, например номер рейса;
- глаголы, например размещать;
- глагольные группы, например проверить действительность кредитной карточки.

Существительные и именные группы могут служить признаком классов или атрибутов классов. Глаголы и глагольные группы – обязанностей классов.

Если встретился термин, значение которого осталось непонятным, необходимо незамедлительно проконсультироваться у эксперта в предметной области и добавить этот термин в глоссарий проекта. Составьте список существительных, именных групп, глаголов и глагольных групп, используйте глоссарий проекта, чтобы разрешить противоречия с синонимами и омонимами. Результатом этого должен стать список потенциальных классов, атрибутов и обязанностей.

После создания такого списка производится предварительное распределение атрибутов и обязанностей по классам. Сделать это можно, вводя классы в средство моделирования и добавляя в них обязанности в качестве операций. Если были выявлены какие-либо предполагаемые атрибуты, их также можно предварительно распределить по классам. В ходе этого процесса может возникнуть некоторое представление об отношениях между определенными классами (прецеденты – хорошие источники этого), поэтому можно добавить и кое-какие потенциальные ассоциации. В результате получается модель классов в первом приближении, которая будет уточняться в ходе дальнейшего анализа.

8.4.2. Выявление классов с помощью CRC-анализа

Очень хорошим (и забавным) способом привлечения пользователей к процессу выявления классов является применение CRC-анализа (CRC – class-responsibilities-collaborators, класс-обязанности-участники). Этот технический прием использует самый мощный в мире инст-

румент анализа – клеящуюся записку (стикер)! CRC-метод настолько популярен, что говорят (может быть, это выдумки), что в какой-то момент компания, торгующая стикерами, стала выпускать их уже с разметкой для имени класса, обязанностей и участников.

> CRC – это техника мозгового штурма, при которой важные моменты предметной области записываются на стикерах.

Начинаем с разметки стикеров, как показано на рис. 8.4. Записка делится на три ячейки. В верхней записывается имя предполагаемого класса, в левой – обязанности, в правой – участники. Участники – это другие классы, которые могут взаимодействовать с этим классом для реализации функциональности системы. Ячейка с участниками обеспечивает возможность записи отношений между классами. Другой способ показать отношения (который мы считаем предпочтительным) – приклеить записки на доску и провести линии между взаимодействующими классами.

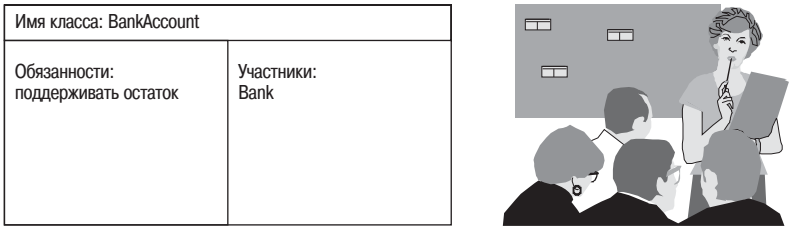


Рис. 8.4. *Разметка стикера*

8.4.2.1. Процедура CRC-анализа

Если это не совсем простая система, CRC-анализ должен всегда использоваться в сочетании с анализом существительное/глагол прецедентов, требований, глоссария и другой относящейся к делу документации. Процедура CRC-анализа проста. Ее основная идея – рассортировать данные, *поступающие* в результате *анализа* информации. По этой причине CRC-анализ лучше проводить в два этапа.

8.4.2.2. Этап 1: мозговой штурм – сбор информации

> Привлечение заинтересованных сторон – важнейшая составляющая успеха CRC.

Участники – ОО аналитики, заинтересованные стороны и эксперты. Кроме того, необходим руководитель. Процедура первого этапа сводится к следующему:

1. Объясните участникам, что это настоящий мозговой штурм.

1.1. Все идеи принимаются как хорошие.

1.2. Идеи записываются, но *не* обсуждаются – никаких споров, просто записывайте идеи и двигайтесь дальше. Все будет анализироваться позже.

2. Попросите членов команды назвать «сущности» их области деятельности, например покупатель, продукт.

2.1. Каждую сущность запишите на клеящуюся записку в качестве предполагаемого класса или атрибута класса.

2.2. Приклейте записку на стену или доску.

3. Попросите команду сформулировать обязанности этих сущностей, запишите их в ячейке обязанностей записки.

4. Работая с командой, попытайтесь установить классы, которые могут работать совместно. Перегруппируйте записки на доске соответственно такой организации классов и нарисуйте линии между ними. В качестве альтернативы впишите имена участников в соответствующую ячейку записки.

8.4.2.3. Этап 2: анализ информации

> Важные бизнес-понятия обычно становятся классами.

Участники – ОО аналитики и эксперты. Как определить, какие из клеящихся записок должны стать классами, а какие – атрибутами? Вернемся к разделу 8.3.2: классы анализа *должны* представлять четкую абстракцию предметной области. Определенные клеящиеся записки будут представлять ключевые бизнес-понятия и, непременно, должны стать классами. Другие записки могут стать классами или атрибутами. Если по логике кажется, что записка является *частью* другой записки, то это верный признак того, что она представляет атрибут. Кроме того, если записка не кажется особенно важной или не отличается интересным поведением, стоит рассмотреть возможность сделать ее атрибутом другого класса.

Если по поводу записки есть какие-то сомнения, просто сделайте ее классом. Важно сделать лучшее приближение, а затем довести этот процесс до логического конца. Всегда можно уточнить модель позже.

8.4.3. Выявление классов путем применения стереотипов RUP

Из RUP был заимствован метод стереотипов. Идея этого метода состоит в том, что в ходе анализа рассматриваются три разных типа класса анализа. Этот метод позволяет сосредоточить анализ на конкретных аспектах системы. Мы считаем необязательным использование этого метода. Его можно применять как дополнение к основным методам анализа: существительное/глагол и CRC-карточки.

> Согласно RUP считается полезным поискать классы, которые можно обозначить стереотипами **«boundary»** (граница), **«control»** (управление) и **«entity»** (сущность).

Стереотипами, приведенными в табл. 8.1, могут быть обозначены три типа классов анализа.

В следующих трех разделах рассматриваются способы выявления каждого из этих типов классов анализа.

Таблица 8.1

Стереотип	Пиктограмма	Семантика
«boundary»		Класс, который является посредником во взаимодействии между системой и ее окружением.
«control»		Класс, инкапсулирующий характерное для прецедента поведение.
«entity»		Класс, используемый для моделирования постоянной информации о чем-то.

8.4.3.1. Выявление классов типа «boundary»

Классы типа «boundary» существуют на границе системы и общаются с внешними актерами.

Такие классы можно обнаружить, рассмотрев контекст системы (границы системы) и выяснив, какие классы являются посредниками между контекстом системы и его окружением. Согласно RUP существует три типа «boundary» (граничных) классов:

1. Классы пользовательского интерфейса – классы, связывающие систему и людей.
2. Классы системного интерфейса – классы, связывающие систему с другими системами.
3. Классы аппаратного интерфейса – классы, связывающие систему с внешними устройствами, например датчиками.

Каждая связь между актером и прецедентом в модели должна быть представлена некоторым объектом системы. Эти объекты – экземпляры граничных классов. Выяснив, кого или что представляет актер (табл. 8.2), можно определить тип граничного класса.

Если граничный класс обслуживает более одного актера, эти актеры должны быть одного типа (представлять человека, систему или устройство). Если граничный класс обслуживает актеров разных типов, это свидетельствует о плохом проектировании!

Таблица 8.2

Актер	Указывает на
Представляет человека	Класс пользовательского интерфейса (GUI)
Представляет систему	Класс системного интерфейса
Представляет устройство	Класс аппаратного интерфейса

Поскольку речь идет о фазе анализа, важно удерживать эти классы на соответствующем уровне абстракции. Например, при моделировании граничного класса, представляющего GUI, необходимо смоделировать только главное окно. Детализация всех графических элементов, составляющих окно, должна быть перенесена в фазу проектирования. Или можно ввести пустой класс, представляющий весь пользовательский интерфейс.

Аналогично и с классами системного и аппаратного интерфейсов. Главное – зафиксировать факт существования класса-посредника между системой и чем-то еще, но *не* конкретные детали этого класса. Детали будут рассматриваться позже, при проектировании.

Например, если создается система электронной коммерции, нуждающаяся во взаимодействии с системой Inventory (инвентаризация), то интерфейс для связи с этой системой можно представить классом InventoryInterface, помеченным стереотипом «boundary». Такой детализации достаточно для аналитической модели.

8.4.3.2. Выявление классов типа «control»

Эти классы являются управляющими – их экземпляры координируют поведение системы, соответствующее одному или более прецедентам.

Управляющие классы выявляются при рассмотрении поведения системы, описанного прецедентами, и выработке решения о том, как это поведение должно быть разделено между классами анализа. Простое поведение часто можно распределить между граничными классами или классами-сущностями. Но более сложное поведение, такое как обработка заказов, лучше обозначить, добавив соответствующий управляющий класс, например OrderManager (менеджер заказов). Некоторые разработчики моделей (включая и нас!) часто обозначают управляющий класс, дополняя его имя словами Manager или Controller.

Главное при работе с управляющими классами – они должны естественным образом проистекать из самой предметной области. Некоторые разработчики моделей искусственно вводят управляющие классы для каждого прецедента, чтобы управлять или выполнять этот прецедент. Это опасный подход, в результате которого получается модель, больше похожая на прямую функциональную декомпозицию, чем на настоящую ОО аналитическую модель. По сути, это одна из причин, по кото-

рой мы считаем использование стереотипов RUP необязательным. Они могут сбить с толку начинающих разработчиков моделей!

В реальности контроллеры, вытекающие непосредственно из предметной области (а не появляющиеся как побочный продукт определенной техники анализа), обычно охватывают несколько прецедентов. Удачным примером контроллера является класс Registrar (регистратор). Он задействован во многих прецедентах, описывающих систему регистрации курса. Аналогично одному прецеденту может требоваться участие нескольких управляющих классов.

Если управляющий класс обладает очень сложным поведением, это говорит о том, что, вероятно, его можно разбить на два или более простых контроллера, реализующих связную часть этого поведения. Каждый из получившихся простых классов должен по-прежнему представлять что-то, что реально имеет место в предметной области. Например, при проектировании системы регистрации курса сначала может быть введен управляющий класс CourseRegistrationController (контроллер регистрации курса), координирующий весь процесс. Но поведение такого класса очень сложное, поэтому его можно разбить на ряд взаимодействующих классов, каждый из которых будет отвечать за один-два аспекта этого поведения. Класс можно было бы разложить на классы Registrar, CourseManager (менеджер курсов) и PersonnelManager (менеджер персонала). Обратите внимание, что каждый из этих классов представляет реальную сущность предметной области.

Хороший способ проанализировать управляющие классы – представить себя в роли такого класса. Что вам пришлось бы сделать в такой ситуации?

8.4.3.3. Выявление классов типа «entity»

Эти классы моделируют информацию о чем-то и обычно имеют очень простое поведение, состоящее практически только в получении и задании значений. Классы, представляющие постоянную информацию, например адреса (класс Address) и людей (класс Person), являются классами-сущностями.

Классы-сущности:

- пересекают несколько прецедентов;
- с ними оперируют управляющие классы;
- предоставляют и принимают информацию от граничных классов;
- представляют ключевые сущности, которыми управляет система (например, Customer);
- часто являются постоянными (persistent).

Классы-сущности выражают логическую структуру данных системы. Если есть модель данных, классы-сущности тесно связаны с сущностями или таблицами этой модели.

8.4.4. Выявление классов из других источников

Помимо анализа существительное/глагол, CRC-анализа и стереотипов RUP существует множество других потенциальных источников классов, которые необходимо учитывать. Мы ищем четкие абстракции, которые проецируются на реальные сущности предметной области. Подобным же образом можно найти классы и в реальном мире.

- Все физические объекты, такие как самолет, люди и гостиницы, могут обозначать класс.
- Документооборот – еще один богатый источник классов. Такие вещи, как счета, заказы и сберегательные книжки, могут указывать на возможные классы. Однако при рассмотрении системы документооборота необходимо быть очень осторожными. Во многих компаниях она развивалась в течение многих лет с сохранением поддержки устаревших и неиспользуемых бизнес-процессов, которые новая система, возможно, пытается заменить! Самая неприятная работа для ОО аналитика/проектировщика – автоматизация устаревшей бумажной системы.
- Известные внешние интерфейсы, такие как экраны, клавиатуры, периферийные устройства и другие системы, также могут быть источником предполагаемых классов, особенно в случае встроенных систем.
- Концептуальные сущности – это сущности, важные для функционирования предприятия, но не являющиеся конкретными сущностями. Примером такой сущности может быть LoyaltyProgram (программа лояльности), например призовая карта. Конечно, сама программа не является конкретной сущностью (ее нельзя пощупать!), но это связная абстракция, поэтому ее можно смоделировать как класс.

8.4.4.1. Базовые шаблоны

> Базовые шаблоны могут предоставить готовые компоненты аналитической модели.

В нашей книге «Enterprise Patterns and MDA» [Arlow 1] описывалось понятие, названное нами *базовые шаблоны* (*archetype patterns*). Это шаблоны бизнес-понятий, настолько распространенных в бизнес-системах, что мы решили считать их по-настоящему базовыми. То есть их можно смоделировать один раз и затем использовать повторно, а не моделировать вновь и вновь в каждой новой бизнес-системе. Основная мысль книги состоит в том, что шаблоны можно использовать в исходном или измененном виде, создавая аналитическую модель из *компонентов модели* (*model components*). Эта техника называется *компонентно-ориентированным моделированием* (*component-based modeling*).

Мы предоставляем следующие базовые шаблоны:

- Customer Relationship Management;
- Inventory;
- Money;
- Order;
- Party;
- Party relationship;
- Product;
- Quantity;
- Rule.

Каждый шаблон разработан *очень* подробно и отличается содержательностью. Их использование позволит сохранить массу человекодней или даже человеко-месяцев работы. Если шаблон не вполне подходит для моделируемой системы, он может служить источником полезных идей для написания собственных классов анализа. Это лучше, чем начинать с чистого листа.

Использование шаблонов – это, наверное, самый эффективный способ выявления классов моделей: просто взять их с полки!

8.5. Создание аналитической модели в первом приближении

Чтобы создать аналитическую модель в первом приближении, необходимо в инструментальном средстве моделирования объединить в одну UML-модель результаты анализа существительное/глагол, CRC-анализа, стереотипов RUP и изучения других источников классов (особенно базовых шаблонов). Объединение осуществляется следующим образом:

1. Сравниваются все источники классов.
2. Классы анализа, атрибуты и обязанности, полученные из разных источников, объединяются и вводятся в инструментальное средство моделирования.
   2.1. С помощью глоссария проекта выявляются синонимы и омонимы.
   2.2. Находятся различия в результатах трех методов. Отличия указывают на неопределенности или моменты, требующие более тщательной проработки. Обработайте эти различия сейчас или оставьте это на будущее.
3. Участники (или линии между клеящимися записками на доске) представляют отношения между классами. Моделирование отношений будет обсуждаться в главе 9.

4. Корректируются имена классов, атрибутов и обязанностей согласно какому-либо стандартному соглашению о присваивании имен, принятому в компании, или в соответствии с простыми соглашениями, изложенными в главе 7.

Результатом этой деятельности является набор классов анализа, в котором у каждого класса *может* быть несколько ключевых атрибутов и *должно* быть от трех до пяти обязанностей. Это аналитическая модель в первом приближении.

8.6. Что мы узнали

В этой главе мы описали, что такое классы анализа, и рассмотрели методы поиска таких классов с помощью анализа существительное/глагол, мозгового штурма CRC, применения стереотипов RUP и проверки других источников классов.

Вы узнали следующее:

- В результате UP-деятельности Анализ прецедентов получаем классы анализа и реализации прецедентов.
- Классы анализа представляют четкую, точно определенную абстракцию предметной области.
  - Предметная область – это область, в которой возникла необходимость в программной системе.
  - Классы анализа должны точно и четко проецироваться в реальное бизнес-понятие.
  - Бизнес-понятия часто нуждаются в уточнении во время анализа.
- Аналитическая модель содержит только классы анализа – любые классы, вытекающие из соображений проектирования (области решения), должны быть исключены.
- Классы анализа включают:
  - высокоуровневый набор предполагаемых атрибутов;
  - высокоуровневый набор операций.
- Каковы признаки хорошего класса анализа?
  - Имя класса отражает его назначение.
  - Класс является четкой абстракцией, которая моделирует один конкретный элемент предметной области.
  - Класс проецируется в четко идентифицируемую возможность предметной области.
  - У класса анализа небольшой, определенный набор обязанностей:
    - обязанность – это контракт или обязательство, которое класс имеет перед своими клиентами;

    - обязанность – семантически связный набор операций;
    - класс должен иметь не более трех-пяти обязанностей.
  - Класс имеет высокую внутреннюю связность (cohesion) – все свойства класса служат реализации его назначения.
  - Класс имеет низкую связанность с другими классами (coupling) – для реализации своего назначения класс должен взаимодействовать с небольшим числом классов.
- Каковы признаки плохого класса анализа?
  - Он является функтоидом – классом с одной операцией.
  - Он является всемогущим классом – классом, делающим все. Классам, в именах которых присутствуют слова «system» или «controller», *следует* уделять больше внимания.
  - У него глубокое дерево наследования – в реальности деревья наследования, как правило, небольшие.
  - У него низкая внутренняя связность.
  - У него высокая связанность с другими классами.
- Анализ существительное/глагол:
  - Ищите существительные или именные группы; это потенциальные классы или атрибуты.
  - Ищите глаголы или глагольные группы; это потенциальные обязанности или операции.
  - Процедура: собрать относящуюся к делу информацию и проанализировать ее.
- CRC-анализ – мощная и одновременно забавная техника мозгового штурма.
  - Важные моменты предметной области записываются на клеящихся записках (стикерах).
  - Каждая записка разделена на три ячейки:
    - класс – содержит имя класса;
    - обязанности – содержит список обязанностей этого класса;
    - участники – содержит список других классов, с которыми взаимодействует данный класс.
  - Процедура CRC-анализа – мозговой штурм:
    - попросите членов команды назвать «сущности», которые действуют в их области деятельности, и запишите их на клеящихся записках;
    - попросите команду обозначить обязанности сущностей и запишите их в ячейке обязанностей на записке;
    - попросите команду определить классы, которые могли бы работать совместно, и запишите их в ячейке участников каждой

записки; если стикеры приклеены на доску, нарисуйте между ними линии.

- Стереотипы RUP могут использоваться с целью сосредоточить анализ на трех типах классов:
  - «boundary» – класс, играющий роль посредника во взаимодействии системы и ее окружения;
  - «control» – класс, инкапсулирующий характерное поведение прецедента;
  - «entity» – класс, используемый для моделирования постоянной информации о чем-то.
- Учтите другие источники классов:
  - физические объекты, документооборот, интерфейсы с внешним миром и концептуальные сущности;
  - базовые шаблоны – ориентированное на компоненты моделирование.
- Создание аналитической модели в первом приближении:
  - сравниваются результаты анализа существительное/глагол, CRC-анализа, применения стереотипов RUP и проверки других источников классов;
  - разрешаются синонимы и омонимы;
  - отличия результатов разных техник указывают на области неопределенности;
  - результаты объединяются в аналитическую модель в первом приближении.

9

Отношения

9.1. План главы

В этой главе рассматриваются отношения между объектами и между классами. Чтобы понять, что такое отношение, прочтите раздел 9.2. Далее обсуждаются: в разделе 9.3 – связи (отношения между объектами), в разделе 9.4 – ассоциации (отношения между классами) и в разделе 9.5 – зависимости (универсальные отношения).

9.2. Что такое отношение?

Отношения – это семантические (значимые) связи между элементами модели. Отношения – это способ объединения сущностей в UML. Нам уже встречались некоторые типы отношений:

- между актерами и прецедентами (ассоциация);
- между прецедентами и прецедентами (обобщение, «include», «extend»);
- между актерами и актерами (обобщение).

> UML-отношения объединяют сущности.

В этой главе рассматриваются взаимоотношения между объектами и между классами. Начнем со связей и ассоциаций, а в главе 10 обсудим обобщение и наследование.

Чтобы создать функциональную ОО систему, нельзя позволять объектам оставаться в гордом одиночестве. Их необходимо объединять, чтобы они могли выполнять полезную для пользователей системы работу. Соединения между объектами называются связями. Когда объекты работают совместно, говорят, что они взаимодействуют.

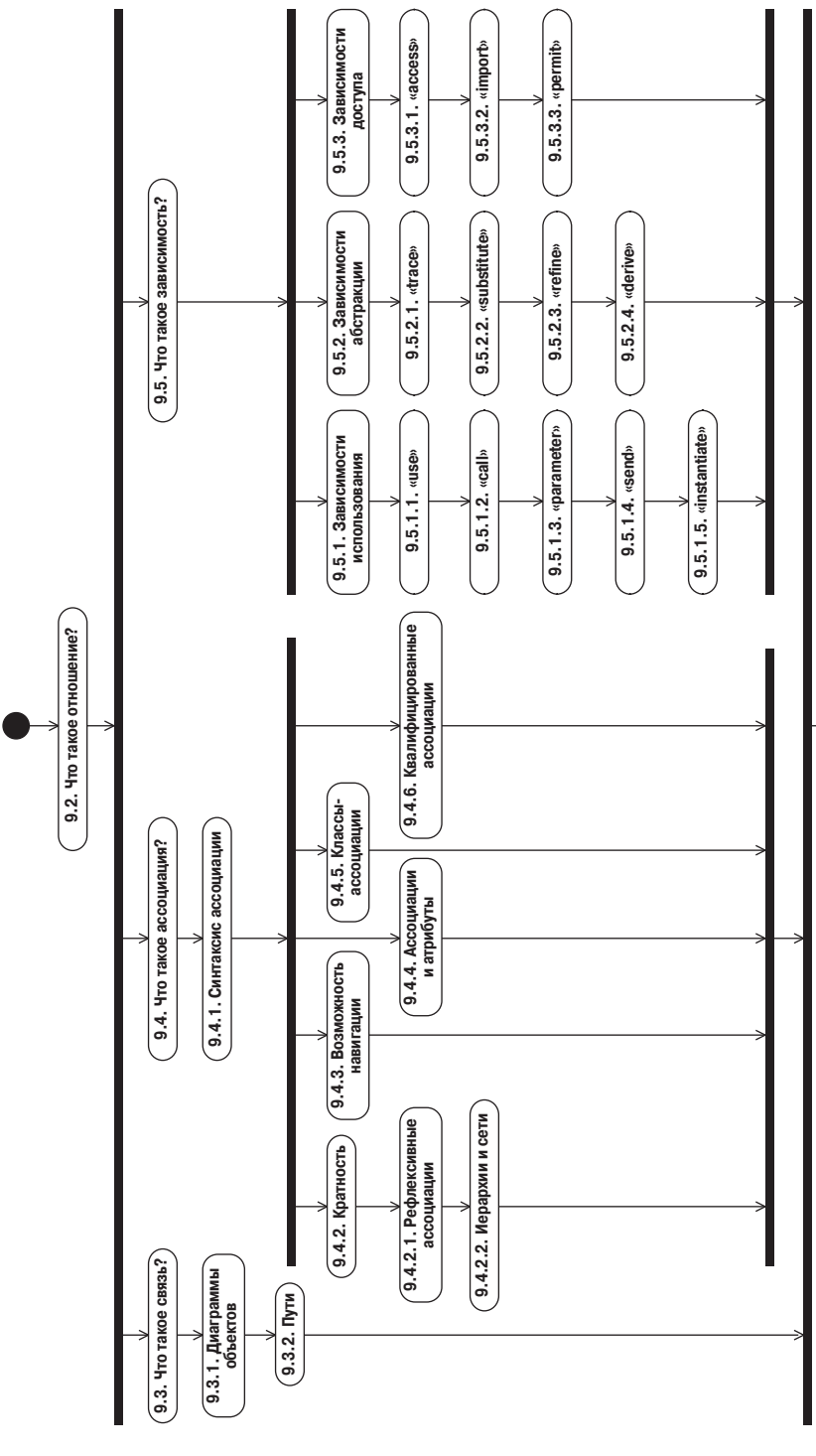


Рис. 9.1. План главы

Если между объектами установлена связь, то и между их классами должно существовать какое-то семантическое соединение. В этом есть здравый смысл: чтобы объекты могли общаться друг с другом напрямую, классы этих объектов должны каким-то образом знать друг о друге. Соединения между классами называются ассоциациями. Связи между объектами на самом деле являются экземплярами ассоциаций между их классами.

9.3. Что такое связь?

Чтобы создать объектно-ориентированную программу, объекты должны общаться друг с другом. В сущности, исполняющаяся ОО программа – это гармоничное сообщество взаимодействующих объектов.

> Объекты обмениваются сообщениями через соединения, называемые связями.

Связь – это семантическое соединение между двумя объектами, которое обеспечивает им возможность обмена сообщениями. Исполняющаяся ОО система содержит множество объектов, которые появляются и исчезают, и множество связей (которые тоже появляются и исчезают), объединяющих эти объекты. Через эти связи происходит обмен сообщениями между объектами. Получив сообщение, объект инициирует соответствующую операцию.

Разные ОО языки программирования реализуют связи по-разному. Java реализует связи как объектные ссылки; C++ может реализовывать связи как указатели, как ссылки или путем прямого включения одного объекта в другой.

Каким бы ни был подход, минимальное требование для установления связи: *по крайней мере один* из объектов должен иметь объектную ссылку на другой.

9.3.1. Диаграммы объектов

Диаграмма объектов – это диаграмма, представляющая объекты и их отношения в некоторый момент времени. Это как снимок части исполняющейся ОО системы в определенный момент, показывающий объекты и связи между ними.

Соединенные связями объекты могут исполнять различные роли по отношению друг к другу. На рис. 9.2 можно видеть, что объект ila играет роль chairperson (председатель) в его связи с объектом bookClub (книжный клуб). На диаграмме объектов это отображается путем размещения имени роли на соответствующем конце связи. Имена ролей могут располагаться на любом или на обоих концах связи. В данном случае объект bookClub всегда играет роль «клуба», поэтому нет особого смысла

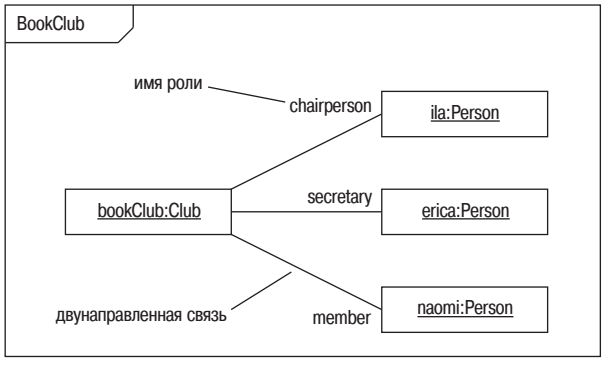


Рис. 9.2. *Двунаправленные связи между объектами*

показывать это на диаграмме. Это никак не улучшило бы понимания отношений объектов.

На рис. 9.2 показано, что в определенный момент времени объект ila играет роль chairperson. Однако важно понимать, что связи – это *динамические соединения* (*dynamic connections*) объектов. Иначе говоря, они могут быть непостоянными во времени. В данном примере роль chairperson может передаваться объектам erica или naomi, и можно было бы без труда создать диаграмму объектов, отображающую эту новую ситуацию.

Обычно одна связь соединяет только два объекта, как показано на рис. 9.2. Однако UML допускает соединение нескольких объектов одной связью. Такую связь называют n-арной. Она изображается в виде ромба, от которого отходят линии к каждому из объектов-участников. Многие разработчики моделей (и мы в том числе) считают такую идиому ненужной. Она редко используется и не всегда поддерживается средствами моделирования UML. Поэтому не будем ее рассматривать.

> Диаграммы объектов – это моментальные снимки работающей ОО системы.

На рис. 9.2 можно найти три связи между четырьмя объектами:

- связь между bookClub и ila;
- связь между bookClub и erica;
- связь между bookClub и naomi.

> Для задания направления движения сообщений по связи используется возможность навигации.

На рис. 9.2 связи двунаправленные, поэтому можно сказать, что связь соединяет ila с bookClub или что связь соединяет bookClub с ila.

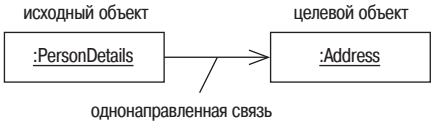


Рис. 9.3. *Однонаправленная связь между объектами*

Если связь однонаправленная, для задания направления движения сообщений по связи используется возможность навигации.

Если навигация возможна, на конце связи помещают стрелку; в противном случае связь завершают крестом (навигация не допускается). Возможность навигации немного похожа на улицу с односторонним движением. Сообщения могут проходить только в том направлении, куда указывает стрелка.

Спецификация UML 2 допускает три разных способа отображения возможности навигации, которые подробно обсуждаются в разделе 9.4.3. В данной книге используется самое распространенное обозначение:

- все кресты опускаются;
- двунаправленные ассоциации изображаются *без* стрелок;
- однонаправленные ассоциации изображаются с одной стрелкой.

Единственным настоящим недостатком этого способа отображения является отсутствие возможности показать, что решение о навигации еще не принято, потому что отсутствие символов навигации подразумевает, что связь «не допускает навигации».

Например, на рис. 9.3 показано, что связь между :PersonDetails и :Address однонаправленная. Это значит, что у объекта :PersonDetails есть объектная ссылка на объект :Address, но *не* наоборот. Сообщения могут пересылаться *только* от :PersonDetails к :Address.

9.3.2. Пути

Обозначения UML, такие как пиктограмма объекта, пиктограмма прецедента и пиктограмма класса, соединяются с другими обозначениями путями. Путь – это «связанные последовательности графических сегментов» (другими словами, линия!), объединяющие два или более обозначений. Существует три стиля представления путей:

- прямоугольный – путь состоит из ряда горизонтальных и вертикальных сегментов;
- наклонный – путь является последовательностью одной или более наклонных линий;
- кривой – путь является кривой линией.

Какой стиль пути использовать, вопрос личных предпочтений. Стили могут даже смешиваться на одной диаграмме, если это делает ее более

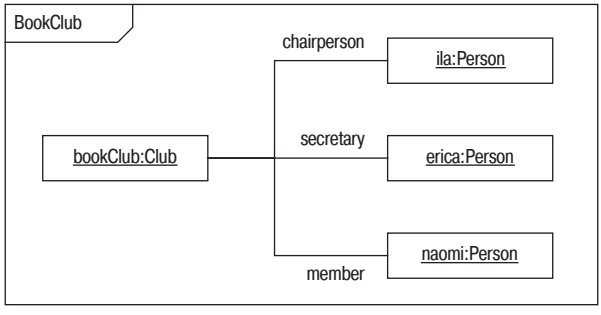


Рис. 9.4. *Прямоугольный стиль отображения путей*

ясной и понятной. Мы, как и многие разработчики моделей, обычно используем прямоугольный стиль.

На рис. 9.4 использован прямоугольный стиль, и пути объединены в дерево. Объединять можно только пути, имеющие одинаковые свойства. В данном случае все пути представляют связи, поэтому вполне законно могут быть объединены.

Визуальная четкость, легкость прочтения и общая привлекательность диаграмм имеют огромное значение. Необходимо всегда помнить, что основная масса диаграмм создается для кого-то. Таким образом, не важно, какой стиль используется, важны четкость и ясность.

9.4. Что такое ассоциация?

Ассоциации – это отношения между классами. Аналогично связям, соединяющим объекты, ассоциации соединяют классы. Самое главное: для того чтобы между двумя объектами была связь, между классами этих объектов *должна* существовать ассоциация. Потому что связь – это экземпляр ассоциации. Так же как объект – экземпляр класса.

> Ассоциации – это соединения между классами.

На рис. 9.5 показаны отношения между классами и объектами и между связями и ассоциациями. Поскольку не может быть связи без ассоциации, очевидно, что связи *зависят* от ассоциаций. Это можно смоделировать как отношение зависимости (пунктирная стрелка), которое более подробно рассматривается в разделе 9.5. Чтобы явно обозначить семантику зависимости между ассоциациями и связями, зависимость помечается стереотипом «instantiate».

> Объекты – это экземпляры классов, а связи – экземпляры ассоциаций.

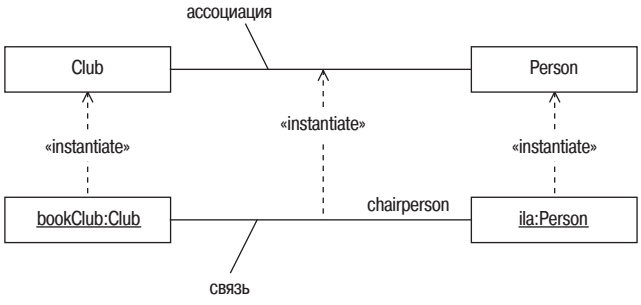


***Рис. 9.5.** Отношения между классами и объектами и между связями и ассоциациями*

Семантика базовой, неуточненной ассоциации чрезвычайно проста: ассоциация между классами указывает на то, что между объектами этих классов могут устанавливаться связи. Существуют другие, более конкретные формы ассоциаций (агрегация и композиция), которые рассматриваются в разделе 18.3 при обсуждении процесса проектирования.

9.4.1. Синтаксис ассоциации

Ассоциации могут иметь:

- имя ассоциации
- имена ролей
- кратность
- возможность навигации

Поскольку ассоциации обозначают действие, производимое исходным объектом над целевым объектом, в качестве их имен должны использоваться глагольные группы. Перед именем или после него может применяться маленькая черная стрелка, указывающая направление, в котором должно читаться имя ассоциации. Имя ассоциации записывается в стиле lowerCamelCase.

В примере на рис. 9.6 ассоциация читается следующим образом: «Компания (Company) нанимает много Человек (Persons)». Хотя стрелка указывает направление чтения ассоциации, всегда можно прочитать ее в обратном направлении. Таким образом, в примере на рис. 9.6 мож-

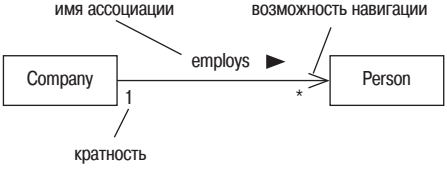


***Рис. 9.6.** Отображение ассоциации с использованием ее имени*

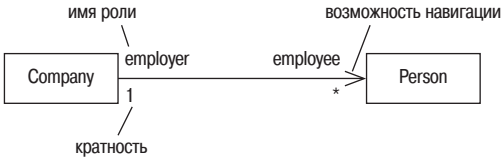


***Рис. 9.7.** Отображение ассоциации с использованием имен ролей*

но сказать, что в любой момент времени «каждый Человек (Person) нанят только одной Компанией (Company)».

> Имена ассоциаций – это глагольные группы, указывающие на семантику ассоциации.

Альтернативный вариант – классам на обоих концах ассоциации могут быть присвоены имена ролей. Эти имена указывают на роли, исполняемые объектами классов, когда они связаны экземплярами данной ассоциации. На рис. 9.7 можно видеть, что объект Company будет играть роль employer (работодатель), а объекты Person – роль employee (служащие), если они связаны экземплярами этой ассоциации. Имена ролей называют роль, которую могут играть объекты, поэтому они должны быть существительными или именными группами.

Ассоциация может иметь *или* имя ассоциации, *или* имя роли. Указание и имени роли, и имени ассоциации для одной и той же ассоциации теоретически допустимо, но считается очень плохим стилем. Это уже перебор!

Признак хороших имен ассоциаций и ролей – они должны легко читаться. На рис. 9.6 ассоциация «Company *employs* many Persons» (Компания *нанимает* много Человек) звучит на самом деле замечательно. Если прочитать ассоциацию в обратном направлении, получается «Person *is employed* by exactly one Company at any point in time» (Человек *нанят* только одной Компанией в любой момент времени). Тоже звучит неплохо. Аналогичным образом имена на рис. 9.7 ясно указывают роли, которые будут играть объекты этих классов, связанные именно таким образом.

> Имена ролей – это именные группы, указывающие на роли, исполняемые объектами, связанными экземплярами этой ассоциации.

9.4.2. Кратность

Ограничения – один из трех механизмов расширения UML, и кратность – первый из рассматриваемых нами типов ограничений. Это также самый распространенный тип ограничений. Кратность ограничивает число объектов класса, которые могут быть вовлечены в конкрет-

***Рис. 9.8.** Отображение кратности на диаграмме*

ное отношение *в любой момент времени*. Фраза «в любой момент времени» – решающая для понимания кратностей. На рис. 9.8 можно увидеть, что в любой момент времени объект Person нанят только одним объектом Company. Однако *с течением времени* объект Person может быть нанят несколькими объектами Company.

На рис. 9.8 можно найти еще кое-какие любопытные вещи. Объект Person не может быть безработным, он всегда нанят только одним объектом Company. Таким образом, ограничение включает в себя два бизнес-правила данной модели:

- объекты Person в данный момент времени могут быть наняты только одним объектом Company;
- объекты Person *не могут* быть безработными.

Не важно, зависят или нет эти допустимые ограничения всецело от требований, предъявляемых к моделируемой системе, но это именно то, что выражает модель.

Как видим, ограничения кратности имеют большое значение, в них могут быть закодированы ключевые бизнес-правила модели. Однако эти правила «скрыты» в деталях модели. Грамотные разработчики моделей называют такое сокрытие ключевых бизнес-правил и требований «тривиализацией» (trivialization). Намного более глубокое обсуждение этого феномена можно найти в книге [Arlow 1].

Кратность задается в виде разделенного запятыми списка интервалов, в котором каждый интервал представлен в форме:

```
минимум..максимум
```

где минимум и максимум – целые числа или любое выражение, возвращающее целое число.

> Если кратность не задана явно, значит, решение о ней еще не принято.

Если кратность не задана явно, значит, решение о ней еще не принято. В UML нет «применяемой по умолчанию» кратности. В моделировании широко распространена ошибка, состоящая в том, что незаданная

кратность по умолчанию предполагается равной 1. Некоторые примеры синтаксиса кратности приведены в табл. 9.1.

> Кратность задает число объектов, которые могут принимать участие в отношении в любой момент времени.

Таблица 9.1

Дополнение	Семантика
0..1	Нуль или 1
1	Ровно 1
0..*	Нуль или более
*	Нуль или более
1..*	1 или более
1..6	От 1 до 6
1..3, 7..10, 15, 19..*	От 1 до 3, *или* от 7 до 10, *или* ровно 15, *или* от 19 до множества

> Модель всегда необходимо читать так, как написано.

Пример на рис. 9.9 иллюстрирует, что кратность действительно является мощным ограничением, которое часто имеет огромное влияние на бизнес-семантику модели.

Если прочитать пример внимательно, можно увидеть, что:

- у Company может быть ровно семь employee;
- Person может быть нанят только одной Company (т. е. в данной модели Person не может одновременно иметь больше одной должности);
- у BankAccount может быть только один owner (владелец);
- у BankAccount может быть один или много operator (операторов);
- у Person может быть от нуля до множества BankAccount;
- Person может управлять от нуля до множества BankAccount.

При прочтении UML-модели крайне важно понять, что именно выражает модель, а не прибегать к предположениям или догадкам. Мы называем это «прочтением модели как есть».

Например, рис. 9.9 утверждает, что у Company может быть именно семь employee, не больше и не меньше. Большинству такая семантика покажется странной или даже неверной (если только это не очень странная компания), но именно так гласит модель. Об этом никогда нельзя забывать.

Существуют некоторые разногласия по поводу того, нужно ли показывать кратность в аналитических моделях. Мы думаем, да, потому что

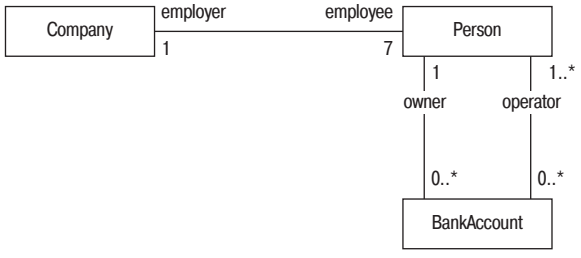


Рис. 9.9. *Кратность – мощное ограничение*

кратность описывает бизнес-правила, требования и ограничения и может вскрыть необоснованные предположения относительно предметной области. Очевидно, что такие предположения необходимо выявлять и устранять как можно раньше.

9.4.2.1. Рефлексивные ассоциации

> Если класс имеет ассоциацию с самим собой, это рефлексивная ассоциация.

Очень распространено явление, когда класс имеет ассоциацию с самим собой. Это называется рефлексивной ассоциацией и означает, что объекты данного класса имеют связи с другими объектами этого же класса. Замечательный пример рефлексивной ассоциации приведен на рис. 9.10. Каждый объект Directory (каталог) может иметь связи с объектами Directory, выступающими в роли subdirectory (подкаталог), число

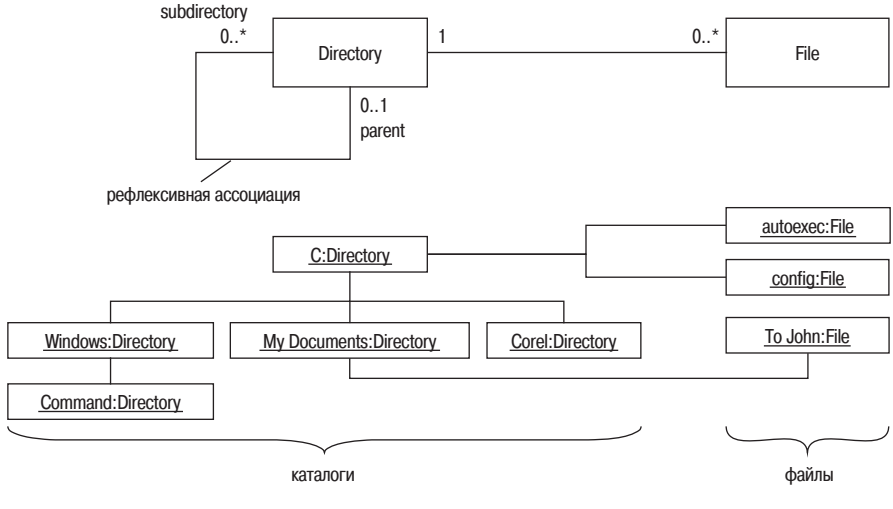


Рис. 9.10. *Пример рефлексивной ассоциации: вверху – диаграмма классов, внизу – диаграмма объектов*

которых может меняться от нуля до некоторой величины (0..*), а также с нулем или одним (0..1) объектом Directory, выступающим в роли parent (родитель). Кроме того, каждый объект Directory ассоциирован с нулем или более объектов File (файл). В этом примере рефлексивные ассоциации превосходно моделируют универсальную структуру каталогов, хотя следует заметить, что у конкретных файловых систем (например, Windows) могут быть другие ограничения кратности.

Верхняя половина рис. 9.10 представляет диаграмму классов. В нижней половине приведен пример диаграммы объектов, соответствующей этой диаграмме классов.

9.4.2.2. Иерархии и сети

В процессе моделирования вы обнаружите, что часто объекты организуются в иерархии или сети. Иерархия имеет один корневой объект. У каждого последующего узла иерархии только один прямой предшественник. Деревья каталогов обычно формируют иерархии. То же самое можно сказать о классификации в машиностроении и о элементах XML- и HTML-документов. Иерархия – это очень упорядоченный, структурированный и довольно негибкий способ организации объектов. Пример иерархии показан на рис. 9.11.

> В иерархии объект может иметь один прямой объект-предок или не иметь ни одного.

Однако в сети обычно нет корневого объекта, хотя это не исключено. В сетях каждый объект может быть непосредственно соединен с многими объектами. В сети нет строгого представления «над» или «под». Это намного более гибкая структура, в которой возможно равенство между узлами. World Wide Web образует сложную сеть узлов, упрощенное представление которой показано на рис. 9.12.

> В сети объект может быть непосредственно соединен с многими объектами или вообще не иметь соединения.

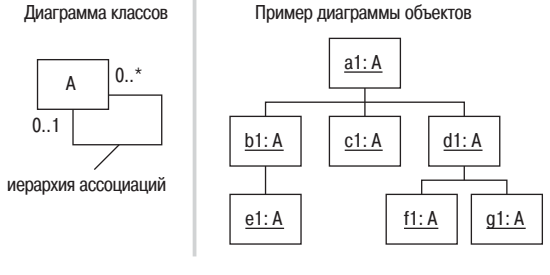


Рис. 9.11. *Пример иерархии*

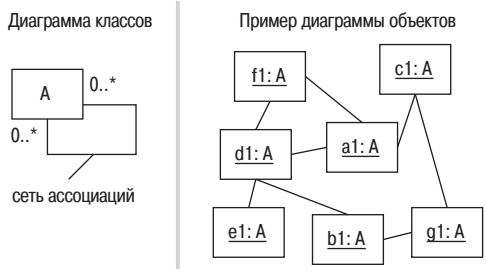


***Рис. 9.12.** Упрощенное представление сети WWW*

Давайте рассмотрим товары в качестве примера, иллюстрирующего иерархии и сети. Существует две фундаментальные абстракции:

- ProductType (тип товара) – тип продукта, например «Струйный принтер»;
- ProductItem (товарная позиция) – конкретный струйный принтер с серийным номером 0001123430.

ProductType и ProductItem очень подробно рассматриваются в книге [Arlow 1]. Типы товаров обычно образуют сети. Таким образом, ProductType, например комплект вычислительного оборудования, может состоять из ЦП, монитора, клавиатуры, мыши, видеокарты и других типов товаров (ProductType). Каждый из этих ProductType описывает *тип* товара, а не конкретную товарную позицию, и эти типы товаров могут входить в другие составные ProductType, например в разные комплекты вычислительного оборудования.

Если же рассматриваются товарные позиции (ProductItem), которые являются конкретными экземплярами ProductType, любая позиция ProductItem (например, конкретный ЦП) может быть продана и поставлена как часть одного комплекта товаров только *один раз*. Следовательно, товарные позиции образуют иерархии.

9.4.3. Возможность навигации

Возможность навигации (navigability) указывает на возможность прохода от объекта исходного класса к одному или более объектам в зависимости от кратности целевого класса. Смысл навигации в том, что «сообщения могут посылаться только в направлении, в котором указывает стрелка». На рис. 9.13 объекты Order могут посылать сообщения объектам Product, но не наоборот.

> Возможность навигации показывает, что объекты исходного класса «знают об» объектах целевого класса.

Одна из целей хорошего ОО анализа и проектирования – минимизировать количество взаимосвязей между классами. И применение возмож-

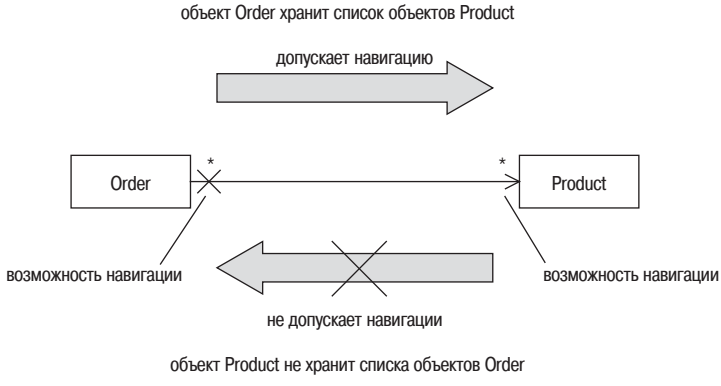


Рис. 9.13. *Варианты обозначения возможности навигации*

ности навигации – верное средство достижения этой цели. Сделав ассоциацию между Order и Product однонаправленной, можно обеспечить возможность свободной навигации от объектов Order к объектам Product, но не в обратном направлении – от объектов Product к объектам Order. Таким образом, объекты Product *не знают* о своем возможном участии в конкретном Order и, следовательно, не имеют связанности с Order.

Возможность навигации обозначается крестом или стрелкой на концах отношения, как показано на рис. 9.13.

Спецификация UML 2.0 [UML2S] предлагает три стиля обозначения возможности навигации на диаграммах модели.

1. Сделать возможность навигации абсолютно явной. Должны быть обозначены все стрелки и кресты.
2. Сделать возможность навигации абсолютно скрытой. Стрелки и кресты не обозначаются.
3. Опускать все кресты. Двунаправленная ассоциация обозначается без стрелок. Однонаправленная ассоциация обозначается с одной стрелкой.

Эти три стиля представлены на рис. 9.14.

Стиль 1 делает возможность навигации полностью видимой, что, однако, может сделать диаграмму слишком громоздкой.

Стиля 2 следует избегать, потому что он скрывает слишком много значимой информации.

Стиль 3 – разумный компромисс. На практике чаще всего используется стиль 3. И поскольку он представляет лучший на данный момент вариант, именно он применяется в этой книге. Основные преимущества стиля 3: при его использовании диаграммы не загромождаются слишком большим количеством стрелок и крестов; он обратно совместим с предыдущими версиями UML. Однако у этого стиля есть и недостатки.

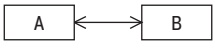
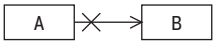
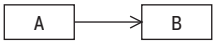
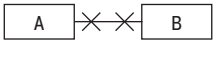
Стили представления возможности навигации в UML 2			
Синтаксис UML 2	Стиль 1: Строгое обозначение возможности навигации в UML 2	Стиль 2: Возможность навигации не обозначается	Стиль 3: Общепринятая практика
	Есть возможность навигации от A к B Есть возможность навигации от B к A		
	Есть возможность навигации от A к B Нет возможности навигации от B к A		
	Есть возможность навигации от A к B Возможность навигации от B к A не определена		Есть возможность навигации от A к B Нет возможности навигации от B к A
	Возможность навигации от A к B не определена Возможность навигации от B к A не определена	Возможность навигации от A к B не определена Возможность навигации от B к A не определена	Есть возможность навигации от A к B Есть возможность навигации от B к A
	Нет возможности навигации от A к B Нет возможности навигации от B к A		

Рис. 9.14. *Стили представления возможности навигации в UML 2*

- Глядя на диаграмму, нельзя сказать, указана ли на ней возможность навигации или она еще не определена.
- Значение связи с одной стрелкой меняется с «есть/не определена» возможность навигации на «есть/нет» возможности навигации. Это недостаток, с которым приходится мириться.
- Нельзя показать ассоциации, в которых отсутствует возможность навигации в обоих направлениях (кресты на обоих концах). Такие связи бесполезны в повседневном моделировании, поэтому особой проблемы это не создает.

Обзор стиля 3 можно увидеть на рис. 9.15.

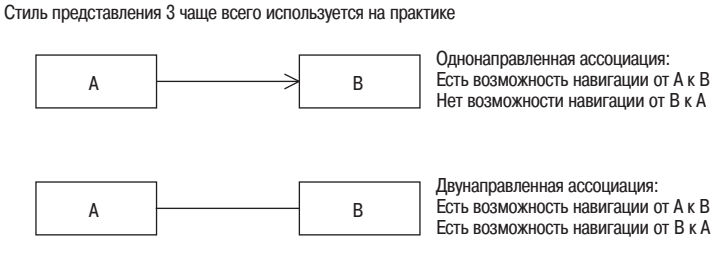


Рис. 9.15. *Стиль 3*

> Даже если нет возможности навигации по ассоциации, все равно может существовать возможность прохода по ассоциации, но затраты на это будут высокими.

Даже если нет возможности навигации по ассоциации, *все равно* может существовать возможность установления отношения в этом направлении. Но вычислительные затраты на это, скорее всего, будут очень высокими. В примере на рис. 9.13, несмотря на отсутствие возможности *прямой* навигации от Product к Order, просмотрев по очереди все объекты Order, можно найти Order, ассоциированный с конкретным объектом Product. То есть можно пройти не допускающее навигации отношение, но с большими вычислительными затратами. Односторонняя возможность навигации подобна улице с односторонним движением: по ней нельзя пойти против движения напролом, но чтобы добраться до ее конца, можно найти другие (более длинные) пути.

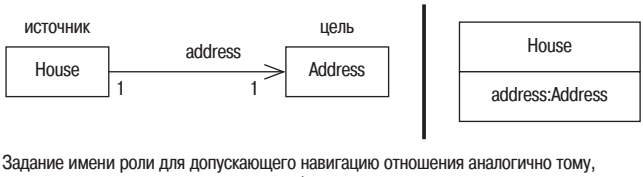
Если на целевом конце отношения указано имя роли, объекты исходного класса могут ссылаться на объекты целевого класса по имени роли.

С точки зрения реализации на ОО языках программирования возможность навигации подразумевает, что у исходного объекта есть объектная ссылка на целевой объект. Исходный объект может использовать эту ссылку для отправки сообщений целевому объекту. На диаграмме объектов это можно представить в виде однонаправленной связи с ассоциированным сообщением.

9.4.4. Ассоциации и атрибуты

Существует тесная связь между ассоциациями класса и атрибутами класса.

Ассоциация между исходным и целевым классами означает, что объекты исходного класса могут сохранять объектную ссылку на объекты целевого класса. Иначе это можно представить так, что ассоциация эквивалентна исходному классу, имеющему псевдоатрибут целевого класса. Объект исходного класса может ссылаться на объект целевого класса с помощью псевдоатрибута (рис. 9.16).



Задание имени роли для допускающего навигацию отношения аналогично тому, что у исходного класса есть псевдоатрибут с тем же именем, что и имя роли, и того же типа, что и целевой класс

***Рис. 9.16.** Объект исходного класса ссылается на объект целевого класса посредством псевдоатрибута*

Нет широко используемого ОО языка программирования, имеющего специальную языковую конструкцию для поддержки ассоциаций. Поэтому при автоматическом генерировании кода из UML-модели ассоциации один-к-одному превращаются в атрибуты исходного класса.

На рис. 9.17 в сгенерированном коде есть класс House с атрибутом address типа Address. Обратите внимание, как имя роли обеспечивает имя атрибуту, а класс, находящийся на другом конце ассоциации, обеспечивает класс атрибута. Приведенный ниже код на Java был сгенерирован из модели, представленной на рис. 9.17:

```
public class House
{
  private Address address;
}
```

Как видите, есть класс House, имеющий один атрибут address типа Address. Обратите внимание, что видимость атрибута address – private; она применяется по умолчанию в большинстве случаев генерации кода.

Если кратность целевого класса больше 1, она реализуется одним из следующих способов:

- как атрибут типа array (массив) (конструкция, поддерживаемая большинством языков программирования);
- как атрибут некоторого типа, являющегося коллекцией.

Коллекции – это просто классы, экземпляры которых имеют особое поведение, заключающееся в способности сохранять и возвращать ссылки на другие объекты. Самым обычным примером коллекции в Java является Vector, но есть и многие другие. Более подробно коллекции обсуждаются в разделе 18.10.

Псевдоатрибуты хороши для отношений один-к-одному и один-ко-многим, но для отношений многие-ко-многим возникают проблемы. Их реализация будет показана в главе 18.

Ассоциации используются только тогда, когда целевой класс является важной частью модели, в противном случае отношения моделируются с помощью атрибутов. Важными являются бизнес-классы, описывающие часть прикладной области. Не имеющими большого значения классами являются компоненты библиотеки, такие как классы String, Date и Time.

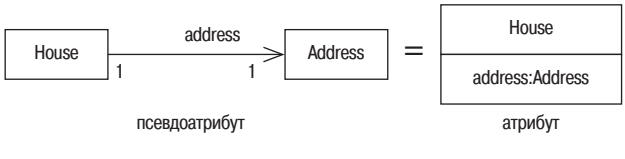


Рис. 9.17. *Из этой модели сгенерирован код, в котором есть класс House с атрибутом address типа Address*

До известной степени выбор явного указания ассоциации или атрибутов является вопросом стиля. Лучшим всегда будет тот подход, в котором модель и диаграммы выражают задачу ясно и точно. Часто представленная на диаграмме ассоциация с другим классом более понятна, чем это же отношение, смоделированное как атрибут, который заметить намного сложнее. Если кратность цели больше 1, это ясно показывает, что целевой класс важен для модели. Таким образом, для моделирования такого отношения используются ассоциации.

Если кратность целевого класса равна 1, целевой объект может быть только частью исходного, и поэтому не стоит показывать отношение с ним как ассоциацию. Возможно, лучше смоделировать его как атрибут. Это особенно справедливо, если кратность равна 1 на *обоих* концах отношения (как на рис. 9.17), когда ни источник, ни цель не могут существовать самостоятельно.

9.4.5. Классы-ассоциации

В ОО моделировании распространена следующая проблема: когда между классами установлено отношение многие-ко-многим, встречаются такие атрибуты, которые не удается поместить ни в один из классов. Проиллюстрируем это примером, приведенным на рис. 9.18.

На первый взгляд это довольно безобидная модель:

- каждый человек (объект Person) может работать во многих компаниях (объект Company);
- каждая компания (Company) может нанимать много людей (объект Person).

Однако что происходит, если добавить бизнес-правило, заключающееся в том, что каждый Person получает зарплату в каждой нанявшей его Company? Где должна быть записана эта зарплата: в классе Person или в классе Company?

Действительно, нельзя сделать зарплату Person атрибутом класса Person, потому что каждый экземпляр Person может работать на многие Company и в каждой получать разную зарплату. Аналогично нельзя сделать зарплату атрибутом Company, поскольку каждый экземпляр Company нанимает множество Person, зарплата которых может быть разной.

Решение кроется в том, что зарплата на самом деле является *собственностью самой ассоциации*. У каждой ассоциации найма, устанавливаемой между объектом Person и объектом Company, своя индивидуальная зарплата.

***Рис. 9.18.** Отношение многие-ко-многим*

UML позволяет моделировать эту ситуацию с помощью класса-ассоциации (рис. 9.19). Важно понимать этот синтаксис: многие люди думают, что класс-ассоциация – это всего лишь прямоугольник, свисающий с ассоциации. Но ничто не могло бы быть дальше от истины, чем это. На самом деле класс-ассоциация – это линия ассоциации (включая все имена ролей и кратности), пунктирная нисходящая линия и прямоугольник класса на конце пунктирной линии. Короче говоря, класс-ассоциация – все, что входит в затемненную область (рис. 9.19).

> Класс-ассоциация – это ассоциация, являющаяся еще и классом.

По сути, класс-ассоциация – это ассоциация, являющаяся *еще* и классом. Она не только соединяет два класса, как обычная ассоциация, она определяет набор характеристик, принадлежащих самой ассоциации. У классов-ассоциаций могут быть атрибуты, операции и другие ассоциации.

> Класс-ассоциация означает, что в любой момент времени между любыми двумя объектами может существовать только одна связь.

Экземпляры класса-ассоциации – это на самом деле *связи*, у которых есть атрибуты и операции. Уникальная идентификация этих связей определяется *исключительно* индивидуальностью объектов, находящихся на каждом конце. Этот фактор ограничивает семантику класса-ассоциации: его можно использовать только тогда, когда между двумя объектами в любой момент времени установлена *единственная уникальная связь*. Это обусловлено тем, что каждая связь, которая является экземпляром класса-ассоциации, должна быть уникальной. На рис. 9.19 применение класса-ассоциации означает, что на модель накладывается следующее ограничение: для данного объекта Person и данного объекта Company может существовать только *один* объект Job (должность). Иначе говоря, каждый Person может занимать только одну Job в данной Company.

Однако если ситуация такова, что данный объект Person может занимать несколько Job в данном объекте Company, класс-ассоциацию использовать нельзя – семантика не соответствует!

Но нам по-прежнему надо куда-то сохранять зарплату для каждой группы Company/Job/Person. Поэтому мы материализуем (делаем реаль-

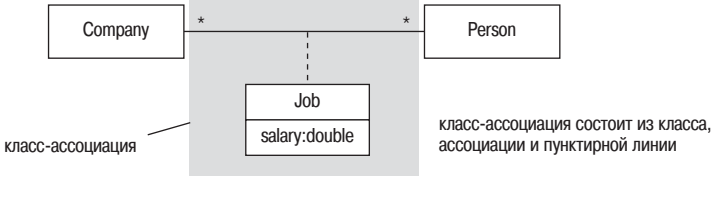


Рис. 9.19. *Класс-ассоциация*

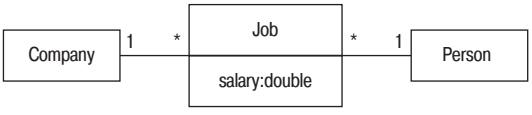


Рис. 9.20. *Материализованное отношение в виде обычного класса*

ным) отношение, представляя его в виде обычного класса. На рис. 9.20 Job является обычным классом. Как видите, у Person может быть несколько Job, каждая Job в каждой конкретной Company.

> Материализованное отношение делает возможным существование более одной связи между любыми двумя объектами в конкретный момент времени.

Откровенно говоря, многие разработчики объектных моделей просто не понимают семантической разницы между классами-ассоциациями и материализованными отношениями. Поэтому одни часто подменяются другими. Однако разница на самом деле очевидна: классы-ассоциации могут использоваться только в том случае, когда каждая связь имеет уникальную индивидуальность. Надо просто запомнить, что индивидуальность связи определяется индивидуальностью объектов, располагающихся на ее концах.

9.4.6. Квалифицированные ассоциации

Квалифицированные ассоциации могут использоваться для превращения ассоциации n-ко-многим в ассоциацию n-к-одному путем задания одного объекта (или группы объектов) из целевого набора. Это очень полезные элементы модели, поскольку они показывают, как можно вести поиск или осуществлять навигацию к конкретным объектам коллекции.

Рассмотрим модель, изображенную на рис. 9.21. Объект Club (клуб) связан с набором объектов Member (член), а объект Member подобным же образом связан с только одним объектом Club.

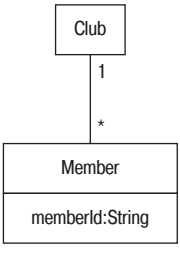


Рис. 9.21. *Отношение один-ко-многим*

> Квалифицированная ассоциация выбирает один объект из целевого набора.

Возникает следующий вопрос: как от объекта Club, связанного с набором объектов Member, перейти к одному конкретному объекту Member? Очевидно, что необходим некоторый уникальный ключ, который можно использовать для поиска определенного объекта Member из набора. Такой ключ называют квалификатором (qualifier). Квалификаторы могут быть разными (имя, номер кредитной карточки, номер социальной страховки). В приведенном выше примере у каждого объекта Member есть значение атрибута memberId, уникальное для данного объекта. Это и есть ключ поиска в данной модели.

В модели такой поиск можно обозначить путем добавления квалификатора на конце ассоциации со стороны Club. Важно понимать, что этот квалификатор принадлежит *концу ассоциации*, а *не* классу Club. Этот квалификатор задает уникальный ключ и таким образом превращает отношение один-ко-многим в отношение один-к-одному, как показано на рис. 9.22.

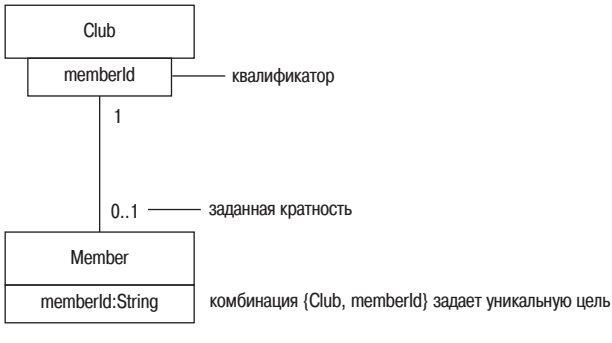


Рис. 9.22. *Квалификатор превращает отношение один-ко-многим в отношение один-к-одному*

Квалифицированные ассоциации – замечательный способ показать, как с помощью уникального ключа происходит выбор определенного объекта из набора. Квалификаторы *обычно* ссылаются на атрибут целевого класса, но возможны и другие выражения, с помощью которых выбирается один объект из набора.

9.5. Что такое зависимость?

Зависимость обозначает отношение между двумя или более элементами модели, при котором изменение одного элемента (поставщика) может повлиять или предоставить информацию, необходимую другому элементу (клиенту). Иначе говоря, клиент некоторым образом зависит

от поставщика. Зависимости используются для моделирования отношений между классификаторами, когда один классификатор зависит от другого, но отношение не является ни ассоциацией, ни обобщением.

> В отношении зависимости клиент некоторым образом зависит от поставщика.

Например, объект одного класса передается как параметр в операцию объекта другого класса. Очевидно, что между классами этих объектов существует отношение некоторого рода, но это не совсем обычная ассоциация. Отношение зависимости (специализированное некоторыми предопределенными стереотипами) можно использовать как универсальное средство для моделирования данного вида отношений. Мы уже приводили один из типов зависимостей – отношение «instantiate», но существуют и многие другие. Общепринятые стереотипы зависимостей рассматриваются в следующих разделах.

UML 2 определяет три основных типа зависимостей (табл. 9.2). Мы включили их в обсуждение для полноты информации, но в повседневном моделировании редко используется что-либо кроме простой пунктирной стрелки зависимости. Обычно разработчики моделей не утруждают себя определением типа зависимости.

Таблица 9.2

Тип	Семантика
Usage (Использование)	Клиент использует некоторые из доступных сервисов поставщика для реализации собственного поведения; это самый распространенный тип зависимости.
Abstraction (Абстракция)	Обозначает отношение между клиентом и поставщиком, где поставщик более абстрактный, чем клиент. Что подразумевается под «более абстрактный»? Это может означать, что поставщик находится на другой стадии разработки, чем клиент (например, в аналитической модели, а не в проектной модели).
Permission (Доступ)	Поставщик предоставляет клиенту разрешение на доступ к своему содержимому – это дает возможность поставщику контролировать и ограничивать доступ к своему содержимому.

Зависимости также могут существовать не только между классами. Как правило, они могут устанавливаться между:

- пакетами и пакетами;
- объектами и классами.

Зависимости могут устанавливаться между операцией и классом. Но эти отношения довольно редко отображаются на диаграммах, поскольку это приводит к слишком большому уровню детализации. Не-

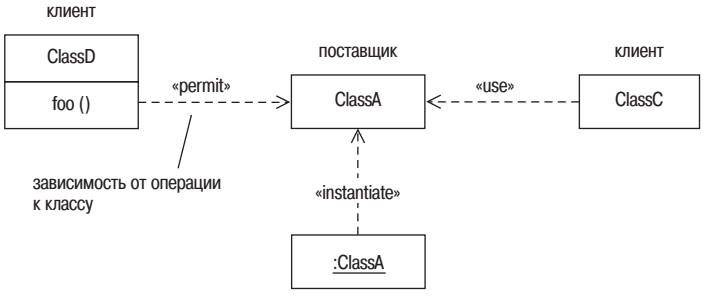


Рис. 9.23. *Типы зависимостей*

которые примеры различных типов зависимостей приведены на рис. 9.23. Они обсуждаются в оставшихся разделах данной главы.

Чаще всего для обозначения зависимости используется обычная пунктирная линия со стрелкой без указания типа зависимости. По сути, тип зависимости часто понятен и без стереотипа – из контекста. Если же возникает желание или необходимость конкретизировать тип зависимости, UML определяет целый ряд стандартных стереотипов.

9.5.1. Зависимости использования

Существует пять зависимостей использования: «use», «call», «parameter», «send» и «instantiate», каждая из которых рассматривается в следующих подразделах.

9.5.1.1. Зависимость «use»

Самым распространенным стереотипом зависимости является «use», который просто обозначает, что клиент каким-то образом использует поставщика. Если на диаграмме указана просто пунктирная линия со стрелкой зависимости без стереотипа, можно быть совершенно уверенным, что подразумевается зависимость «use».

На рис. 9.24 показаны два класса, A и B, между которыми установлено отношение зависимости «use». Эта зависимость генерируется любой из следующих ситуаций.

1. Операции класса A необходим параметр класса B.
2. Операция класса A возвращает значение класса B.

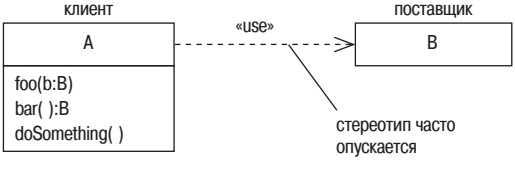


Рис. 9.24. *Два класса с отношением зависимости типа «use»*

3. Операция класса A где-то в своей реализации использует объект класса B, но не в качестве атрибута.

Варианты 1 и 2 довольно просты, а вот вариант 3 представляет больший интерес. Такая ситуация возможна, если одна из операций класса A создала временный объект класса B. Ниже приводится фрагмент Java-кода для этого случая:

```
class A
{
  …
  void doSomething()
    {
      B myB = new B();
      // используем myB некоторым образом
      …
    }
}
```

Хотя одна зависимость «use» может использоваться как универсальная для всех трех перечисленных случаев, есть и другие, более специализированные стереотипы зависимостей, которые можно было бы применить.

Ситуации 1 и 2 можно более точно смоделировать с помощью зависимости «parameter», а ситуацию 3 – с помощью зависимости «call». Однако от UML-модели не часто требуется такой уровень детализации, и большинство разработчиков моделей считают, что намного понятней и проще просто устанавливать между соответствующими классами зависимость «use», как показано выше.

9.5.1.2. Зависимость «call»

Зависимость «call» (вызов) устанавливается между операциями – операция-клиент вызывает операцию-поставщик. Этот тип зависимости в UML-моделировании используется не очень широко. Он применяется на уровне детализации, более глубоком, чем тот, на который большинство разработчиков готовы пойти. Кроме того, в настоящее время очень немногие средства моделирования поддерживают зависимости между операциями.

9.5.1.3. Зависимость «parameter»

Поставщик является параметром операции клиента.

9.5.1.4. Зависимость «send»

Клиент – это операция, посылающая поставщика (который должен быть сигналом) в некоторую неопределенную цель. Сигналы рассматриваются в разделе 15.6, а пока будем представлять их как особые типы классов, используемые для передачи данных между клиентом и целью.

9.5.1.5. Зависимость «instantiate»

Клиент – это экземпляр поставщика.

9.5.2. Зависимости абстракции

Зависимости абстракции моделируют зависимости между сущностями, находящимися на разных уровнях абстракции. В качестве примера можно привести класс аналитической модели и тот же класс в проектной модели. Существует четыре зависимости абстракции: «trace», «substitute», «refine» и «derive».

9.5.2.1. Зависимость «trace»

Зависимость «trace» часто используется, чтобы проиллюстрировать отношение, в котором поставщик и клиент представляют одно понятие, но находятся в разных моделях. Например, поставщик и клиент могут находиться на разных стадиях разработки. Поставщик мог бы быть аналитическим представлением класса, а клиент – более детальным проектным представлением. Также «trace» можно использовать, чтобы показать отношение между функциональным требованием, таким как «банкомат должен обеспечивать возможность снятия наличных денег вплоть до достижения кредитного лимита карты», и прецедентом, поддерживающим это требование.

9.5.2.2. Зависимость «substitute»

Зависимость «substitute» (заместить) показывает, что клиент во время выполнения может заменять поставщика. Замещаемость основывается на общности контрактов и интерфейсов клиента и поставщика, т. е. они должны предоставлять один и тот же набор сервисов. Обратите внимание, что замещаемость не достигается посредством отношений специализации/обобщения между клиентом и поставщиком (специализация/обобщение обсуждаются в разделе 10.2). В сущности, «substitute» специально разработана для использования в средах, не поддерживающих специализации/обобщения.

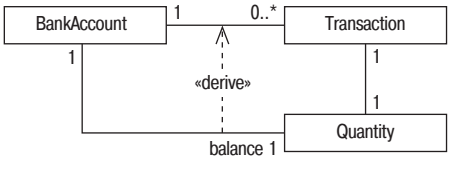
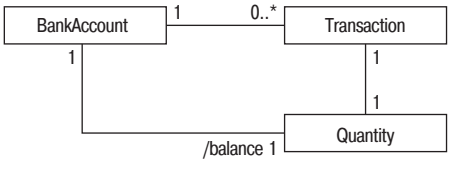
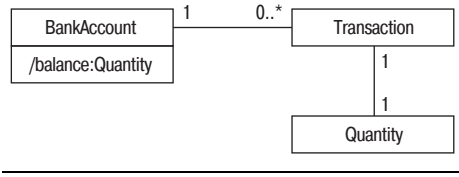
9.5.2.3. Зависимость «refine»

Тогда как зависимость «trace» устанавливается между элементами разных моделей, «refine» (уточнить) может использоваться между элементами одной и той же модели. Например, в модели может быть две версии класса, одна из которых оптимизирована по производительности. Поскольку оптимизация производительности является разновидностью уточнения, это отношение между двумя классами можно смоделировать как зависимость «refine» с примечанием, описывающим суть уточнения.

9.5.2.4. Зависимость «derive»

Стереотип «derive» (получить) используется, когда необходимо явно показать возможность получения одной сущности как производной от другой. Например, в имеющемся классе BankAccount есть список Transaction (транзакция), в котором каждая Transaction содержит Quantity (количество) денег. По требованию всегда можно вычислить текущий баланс, суммируя Quantity по всем Transaction. Существует три способа показать, что balance (баланс) счета (его Quantity) может быть производной сущностью. Они приведены в табл. 9.3.

Таблица 9.3

Модель	Описание
	Класс BankAccount имеет «derive» ассоциацию с Quantity, где Quantity играет роль баланса банковского счета. Эта модель подчеркивает, что balance является производным от коллекции Transaction класса BankAccount.
	В данном случае в имени роли используется слэш, чтобы показать, что между BankAccount и Quantity установлено отношение «derive». Такое обозначение менее явное, поскольку не показывает, производным чего является balance.
	Здесь balance показан как производный атрибут, что обозначено слэшем, предваряющим имя атрибута. Это самое краткое выражение зависимости «derive».

Все перечисленные способы обозначения балансов как производных сущностей эквивалентны, хотя первая модель в табл. 9.3 является наиболее явной. Мы предпочитаем явные модели.

9.5.3. Зависимости доступа

Зависимости доступа выражают способность доступа одной сущности к другой. Существует три зависимости доступа: «access», «import» и «permit».

9.5.3.1. Зависимость «access»

Зависимость «access» (доступ) устанавливается между пакетами. В UML пакеты используются для группировки сущностей. Самое главное

здесь то, что «access» разрешает одному пакету доступ ко всему открытому содержимому другого пакета. Однако каждый пакет определяет пространство имен, и с установлением отношения «access» пространства имен остаются изолированными. Это означает, что элементы клиентского пакета должны использовать имена путей (pathnames), когда хотят обратиться к элементам пакета-поставщика. Более подробное обсуждение данного вопроса представлено в глОднако иве 11.

9.5.3.2. Зависимость «import»

Зависимость «import» концептуально аналогична «access», за исключением того, что пространство имен поставщика объединяется с пространством имен клиента. Это обеспечивает возможность элементам клиента организовывать доступ к элементам поставщика без необходимости указывать в именах элементов имя пакета. Однако иногда это может приводить к конфликтам имен, если имена элемента клиента и элемента поставщика совпадают. Очевидно, что в этом случае необходимо использовать полные имена. В главе 11 этот вопрос обсуждается более подробно.

9.5.3.3. Зависимость «permit»

Зависимость «permit» (разрешить) обеспечивает возможность управляемого нарушения инкапсуляции, но в целом этого отношения следует избегать. Клиентский элемент имеет доступ к элементу-поставщику независимо от объявленной видимости последнего. Часто зависимость «permit» устанавливается между двумя родственными классами, когда клиентскому классу выгодно (вероятно, по причинам производительности) иметь доступ к закрытым членам поставщика. Не все языки программирования поддерживают зависимости «permit». C++ позволяет классу объявлять друзей, которые имеют разрешение на доступ к его закрытым членам. Но эта возможность была (и, наверное, благоразумно) изъята из Java и C#.

9.6. Что мы узнали

В этой главе мы начали обсуждение отношений, которые связывают воедино модели UML. Мы узнали следующее:

- Отношения – это семантические соединения между сущностями.
- Соединения между объектами называются связями.
  - Связь возникает, когда объект сохраняет объектную ссылку на другой объект.
  - Объекты реализуют поведение системы через взаимодействие:
    - взаимодействие возникает, когда объекты обмениваются сообщениями по связям;
    - когда объект получает сообщение, он выполняет соответствующую операцию.

    - Разные ОО языки программирования реализуют связи по-разному.
- Диаграммы объектов отображают объекты и их связи в определенный момент времени.
    - Диаграммы – это моментальные снимки исполняющейся ОО системы в определенный момент времени.
    - Объекты могут играть определенные роли по отношению друг к другу; роль, исполняемая объектом в связи, определяет семантику его участия во взаимодействии.
    - N-арные связи могут объединять более двух объектов; они отображаются в виде ромба, соединенного линиями с каждым объектом, но используются редко.
- Пути – это линии, соединяющие элементы модели UML:
    - прямоугольный стиль – прямые линии, располагающиеся под прямым углом;
    - наклонный стиль – косые линии;
    - криволинейный стиль – кривые линии;
    - необходимо следовать и придерживаться одного стиля, если только сочетание стилей не повышает читаемость диаграммы (что обычно не наблюдается).
- Ассоциации – это семантические соединения между классами.
    - Если между двумя объектами есть связь, между классами этих объектов *должна* существовать ассоциация.
    - Связи – это экземпляры ассоциаций, так же как объекты – экземпляры классов.
    - Ассоциации могут иметь (не обязательно) следующие элементы:
        - Имя ассоциации:
            - перед или после него может стоять маленькая черная стрелка, обозначающая направление чтения этого имени;
            - должно быть глаголом или глагольной группой;
            - записывается в стиле lowerCamelCase;
            - использует или имя ассоциации, или имена ролей, но не одновременно и то, и другое.
        - Имена ролей на одном или обоих концах ассоциации:
            - должны быть существительным или именной группой, описывающими семантику роли;
            - записывается в стиле lowerCamelCase.
        - Кратность:
            - показывает количество объектов, которые могут участвовать в отношении в любой момент времени;

    - объекты могут появляться и исчезать, но кратность ограничивает число объектов, принимающих участие в отношении в любой момент времени;
    - кратность задается разделенным запятыми списком интервалов, например 0..1, 3..5;
    - не существует применяемой по умолчанию кратности – если кратность не обозначена явно, значит, она не определена.
  - Возможность навигации:
    - обозначается стрелкой на конце отношения – если на отношении стрелки не обозначены, значит, оно двунаправленное;
    - возможность навигации показывает, что отношение можно пройти в направлении по стрелке;
    - вероятно, используя другой путь, можно пройти в обратном направлении, но это потребует больших вычислительных затрат.
- Ассоциация между двумя классами эквивалентна наличию у одного класса псевдоатрибута, который может хранить ссылку на объект другого класса:
  - часто ассоциации и атрибуты взаимозаменяемы;
  - ассоциации следует применять, если на одном конце ассоциации находится важный класс и это необходимо подчеркнуть;
  - атрибуты используются, когда класс, участвующий в отношении, не очень важен (например, библиотечные классы, такие как String или Date).
- Класс-ассоциация – это ассоциация, которая является еще и классом:
  - может иметь атрибуты, операции и отношения;
  - класс-ассоциация может использоваться, когда между парой объектов в любой момент времени существует только одна-единственная связь;
  - если пара объектов в определенный момент времени может иметь много связей друг с другом, тогда мы материализуем отношение, заменяя его обычным классом.
- Квалифицированные ассоциации используют квалификатор, чтобы выбрать из целевого набора один объект:
  - квалификатор должен быть уникальным ключом для целевого набора;
  - квалифицированные ассоциации понижают кратность отношения n-ко-многим до n-к-одному;
  - они являются полезным способом обратить внимание на уникальные идентификаторы.

- Зависимости – это отношения, в которых изменение поставщика оказывает влияние или поставляет информацию клиенту.
  - Клиент некоторым образом зависит от поставщика.
  - Зависимости изображаются в виде пунктирной линии со стрелкой, направленной от клиента к поставщику.
  - Зависимости использования:
    - «use» – клиент некоторым образом использует поставщика – универсальная зависимость;
    - «call» – операция клиента инициирует операцию поставщика;
    - «parameter» – поставщик является параметром или возвращаемым значением одной из операций клиента;
    - «send» – клиент отправляет поставщика (который должен быть сигналом) в заданную цель;
    - «instantiate» – клиент является экземпляром поставщика.
  - Зависимости абстракции:
    - «trace» – клиент является историческим развитием поставщика;
    - «substitute» – клиент может замещать поставщика во время выполнения;
    - «refine» – клиент является версией поставщика;
    - «derive» – клиент может быть производным от поставщика:
      - производные отношения могут быть показаны явно с помощью зависимости «derive»;
      - производные отношения могут быть показаны путем добавления слэша перед именем роли или отношения;
      - производные атрибуты можно обозначить путем добавления слэша перед именем атрибута.
  - Зависимости доступа:
    - «access» – зависимость между пакетами, когда клиентский пакет имеет доступ ко всему открытому содержимому пакета-поставщика; пространства имен пакетов остаются изолированными;
    - «import» – зависимость между пакетами, когда клиентский пакет имеет доступ ко всему открытому содержимому пакета-поставщика; пространства имен пакетов объединяются;
    - «permit» – управляемое нарушение инкапсуляции, когда клиент имеет доступ к закрытым членам поставщика; эта зависимость не поддерживается широко, ее следует избегать по мере возможности.

10

Наследование и полиморфизм

10.1. План главы

В этой главе основное внимание уделено ключевым концепциям наследования (раздел 10.3) и полиморфизма (раздел 10.4). Но прежде чем углубиться в эти вопросы, важно понять принцип обобщения, обсуждаемый в разделе 10.2.

10.2. Обобщение

> Обобщение – это отношение между более общей сущностью и более специальной сущностью.

Перед тем как мы сможем перейти к обсуждению наследования и полиморфизма, необходимо сформировать четкое представление об идее обобщения. Обобщение – это отношение между более общим элементом и более специальным элементом, когда более специальный элемент *полностью совместим* с более общим элементом, но содержит большее количество информации.

Два элемента подчиняются принципу замещаемости: более специальный элемент может использоваться *везде*, где предполагается использование более общего элемента, без нарушения системы. Очевидно, что обобщение – намного более прочный тип отношений, чем ассоциация. В самом деле, обобщение подразумевает самый высокий уровень зависимости (и, следовательно, связанности) между двумя элементами.

10.2.1. Обобщение классов

С концептуальной точки зрения идея обобщения проста. Всем хорошо известны общие сущности, например дерево, и более специальные сущности, например дуб, который является конкретным типом дерева.

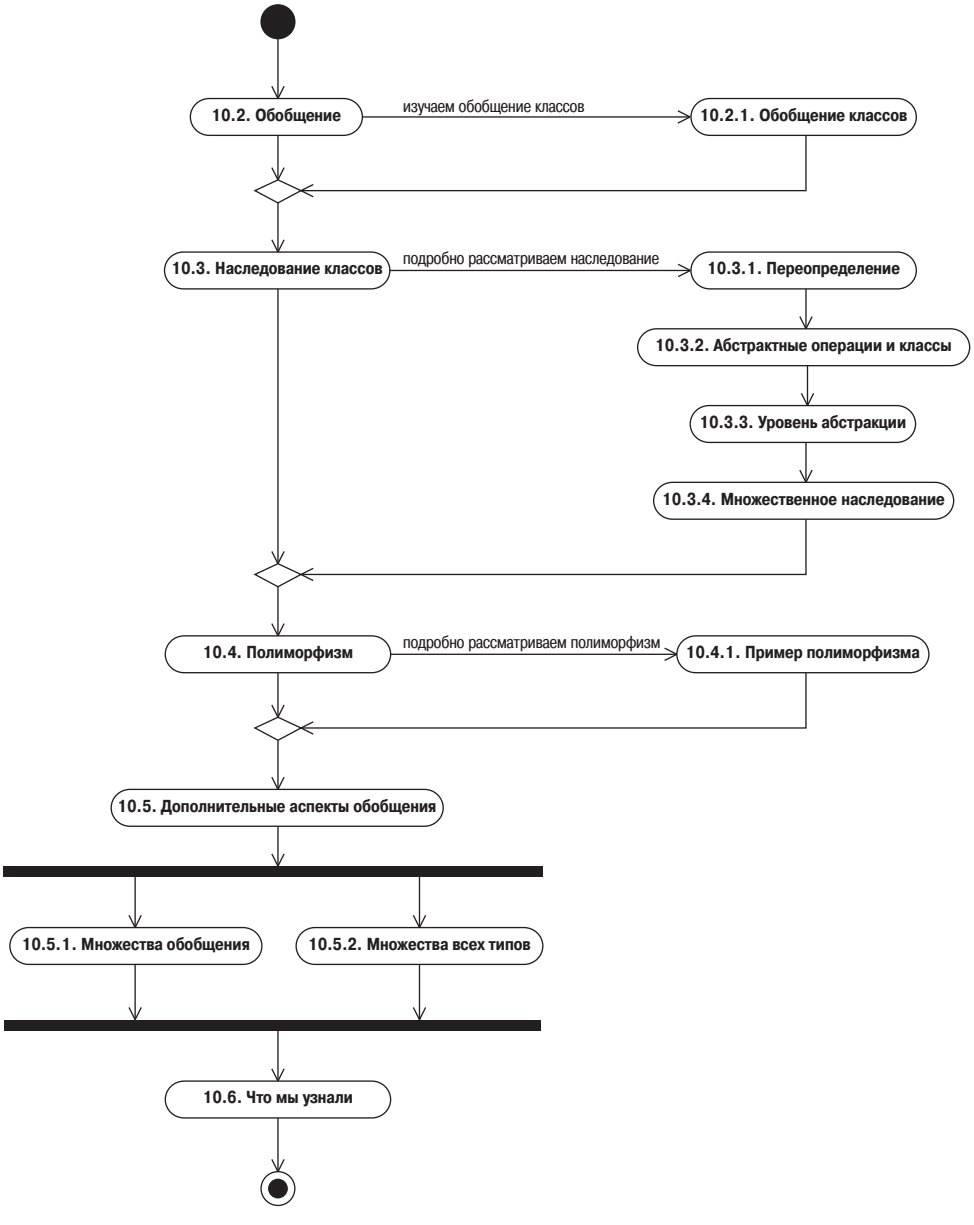


Рис. 10.1. *План главы*

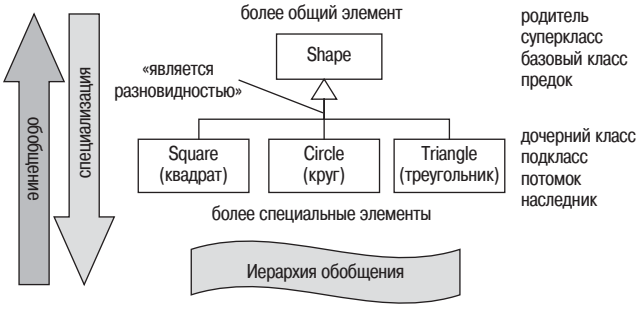


Рис. 10.2. *Иерархия обобщения*

Обобщение применяется ко всем классификаторам. В этой книге нам уже встречалось обобщение, применяемое к прецедентам и актерам. Теперь обсудим его применение к классам.

На рис. 10.2 представлен класс Shape (фигура); это, несомненно, очень общее понятие! От него происходят дочерние классы, подклассы, потомки, наследники (все эти термины широко используются), которые являются более специальными разновидностями общего понятия Shape. Согласно принципу замещаемости экземпляр любого из этих подклассов может использоваться *везде*, где предполагается экземпляр суперкласса (или «надкласса») Shape.

> Иерархия обобщения создается путем обобщения более специальных сущностей и специализации более общих сущностей.

При подробном обсуждении атрибутов и операций этих классов мы увидим, что прийти к приведенной выше иерархии можно двумя способами: либо через процесс специализации, либо через процесс обобщения. В специализации при анализе сначала определяется общая концепция Shape, а затем происходит ее специализация до конкретных типов фигур. В обобщении анализ выявляет более специализированные Square (квадрат), Circle (круг) и Triangle (треугольник), а затем устанавливается, что все они имеют общие характеристики, которые можно выделить в более общий надкласс.

ОО анализ стремится использовать и специализацию, и обобщение одновременно. Хотя из собственного опыта можем сказать, что в процессе анализа следует научиться как можно раньше замечать более общие вещи.

10.3. Наследование классов

В иерархии обобщения (рис. 10.2) кроется наследование между классами, посредством которого подклассы наследуют все возможности

своих надклассов. Чтобы быть более специальными, подклассы наследуют:

- атрибуты;
- операции;
- отношения;
- ограничения.

> Подклассы наследуют характеристики своего суперкласса.

Подклассы также могут вводить новые возможности и переопределять операции суперкласса. Все эти аспекты наследования подробно рассматриваются в следующих разделах.

10.3.1. Переопределение

В примере на рис. 10.3 подклассы Square и Circle класса Shape наследуют все его атрибуты, операции и ограничения. Это означает, что хотя мы и не видим этих элементов в подклассах, они присутствуют в них неявно. Говорят, что Square и Circle типа Shape.

Обратите внимание, что операции draw() (отрисовать) и getArea() (найти площадь), определенные в Shape, не подходят для подклассов. Посылая сообщение draw(), мы ожидаем, что объект Square отрисует квадрат, а объект Circle – круг. Очевидно, что стандартная операция draw(), унаследованная обоими подклассами от их родителя, не годится. Фактически данная операция может вообще ничего не отрисовывать. В конце концов, откуда ей знать, как должен выглядеть Shape? То же самое можно сказать в отношении операции getArea(). Как вычислить площадь Shape?

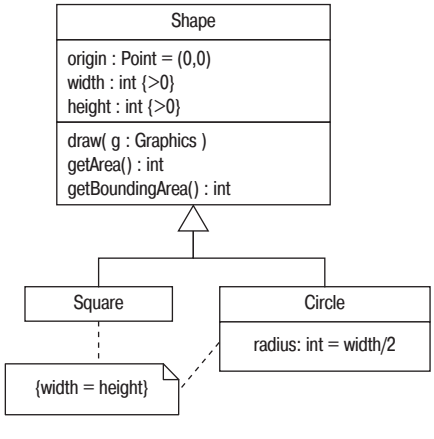


Рис. 10.3. *Наследование характеристик суперкласса*

Эти проблемы явно указывают на необходимость возможности изменения поведения суперкласса в подклассах. Классам Square и Circle надо реализовать собственные операции draw() и getArea(), которые переопределяют стандартные операции, предоставляемые родителем, и обеспечивают более подходящее поведение.

На рис. 10.4 все это показано в действии: подклассы Square и Circle предоставили собственные операции draw() и getArea(), имеющие соответствующее поведение.

- Square::draw(g : Graphics) – отрисовывает квадрат.
- Square::getArea() : int – вычисляет и возвращает площадь квадрата.
- Circle::draw(g : Graphics) – отрисовывает круг.
- Circle::getArea() : int – вычисляет и возвращает площадь круга.

> Подклассы переопределяют унаследованные операции, предоставляя новую операцию с такой же сигнатурой.

Чтобы переопределить операцию надкласса, подкласс должен предоставить операцию с *точно* такой же сигнатурой, что и у переопределяемой операции надкласса. UML определяет сигнатуру операции как имя операции, ее возвращаемый тип и типы всех параметров в порядке перечисления. Имена параметров не учитываются, поскольку они являются просто удобным способом обращения к определенному параметру в теле операции и поэтому не считаются частью сигнатуры.

Все это замечательно, но важно знать, что разные языки программирования могут по-разному определять «сигнатуру операции». Например, в C++ и Java возвращаемый тип операции не является частью сигнатуры операции. Таким образом, если операции подкласса и надкласса будут отличаться только возвращаемым типом, в этих языках будет сформирована ошибка компилятора или интерпретатора.

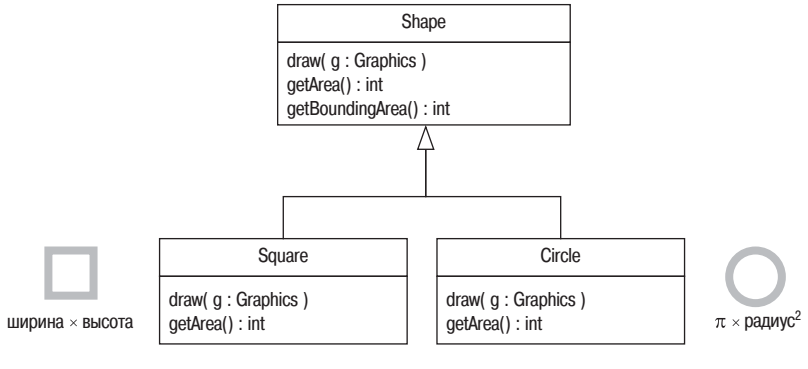


Рис. 10.4. *Переопределение унаследованных операций*

10.3.2. Абстрактные операции и классы

Иногда требуется перенести реализацию операции в подклассы. В примере с классом Shape операция Shape::draw(g : Graphics) – как раз такой случай. В самом классе Shape обеспечить какую-либо разумную реализацию этой операции невозможно, поскольку неизвестно, как должны отрисоваться «фигуры». Понятие «отрисовка фигуры» слишком абстрактное, чтобы иметь конкретную реализацию.

> У абстрактных операций нет реализации.

Отсутствие реализации операции можно обозначить, сделав ее абстрактной операцией. В UML для этого имя операции просто записывается курсивом.

Класс с одной или более абстрактными операциями является неполным, поскольку в нем есть операции, не имеющие реализации. Это означает невозможность создания экземпляров подобных классов. Поэтому такие классы называют абстрактными. Чтобы показать, что класс является абстрактным, его имя записывается курсивом.

> Абстрактные классы имеют одну или более абстрактных операций. Создать экземпляр абстрактного класса невозможно.

В примере на рис. 10.5 абстрактный класс *Shape* имеет две абстрактные операции: *Shape::draw(g : Graphics)* и *Shape::getArea() : int*. Эти операции реализуются подклассами Square и Circle. Хотя *Shape* является неполным, и его экземпляр не может быть создан, оба его подкласса предоставляют недостающие реализации, являются полными и могут иметь экземпляры. Любой класс, экземпляр которого может быть создан, называется конкретным классом.

Операция getBoundingArea() является конкретной операцией класса *Shape*, потому что контактная площадь (bounding area) любой фигуры вычисляется одинаково: ширина фигуры умножается на высоту.

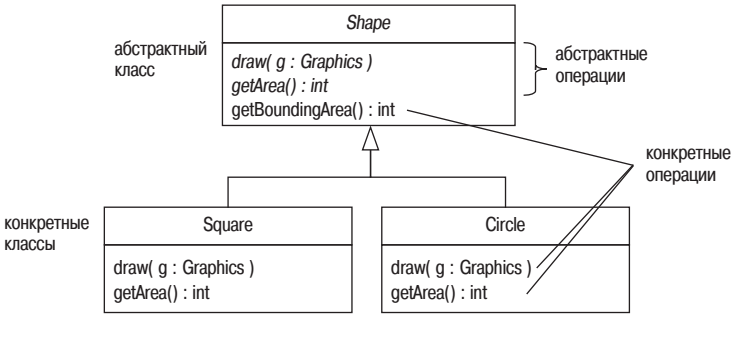


Рис. 10.5. *Абстрактный класс Shape и конкретные подклассы Square и Circle*

Использование абстрактных классов и операций обеспечивает два серьезных преимущества:

- В абстрактном суперклассе можно определять ряд абстрактных операций, которые должны быть реализованы всеми подклассами *Shape*. Это можно рассматривать как определение «контракта», который *должны* реализовать все конкретные подклассы *Shape*.
- Можно написать код управления фигурами и затем подставить Circle, Square и другие подклассы *Shape* соответственно. Согласно принципу замещаемости код, написанный для управления *Shape*, должен работать для всех подклассов *Shape*.

Более подробно эти преимущества рассматриваются при обсуждении полиморфизма в разделе 10.4.

10.3.3. Уровень абстракции

Перед тем как перейти к полиморфизму, неплохо было бы разобраться в уровнях абстракции. Что не так в модели на рис. 10.6?

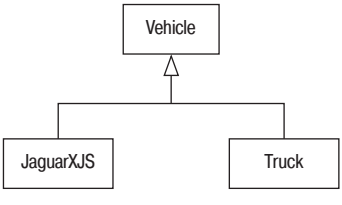


Рис. 10.6. *Пример неверной иерархии*

> Сущности, располагающиеся на одном уровне иерархии обобщения, должны находиться на одном уровне абстракции.

Ответ: «в уровне абстракции». Иерархия обобщения определяет ряд уровней абстракции, начиная от самого общего, находящегося на самом верху, вплоть до самого конкретного, находящегося в самом низу иерархии. Всегда необходимо стараться придерживаться одного уровня абстракции на одном уровне иерархии обобщения. В приведенном выше примере этого *нет*. JaguarXJS – *марка* автомобиля. Очевидно, что это более низкий уровень абстракции, чем Truck (грузовик). Исправить данную модель очень просто. Необходимо ввести между JaguarXJS и Vehicle (транспортное средство) суперкласс Car (автомобиль).

10.3.4. Множественное наследование

> Множественное наследование – у класса может быть более одного непосредственного суперкласса.

UML позволяет классу иметь несколько непосредственных надклассов. Это называется *множественным наследованием* (*multiple inheritance*). Подкласс наследуется от всех его непосредственных надклассов.

Обычно множественное наследование считают вопросом проектирования, поэтому отложим его обсуждение до раздела 17.6.2.

10.4. Полиморфизм

Полиморфизм означает «много форм». Полиморфная операция – это операция, имеющая много реализаций. Мы уже видели две полиморфные операции в примере с классом *Shape*. Абстрактные операции *draw()* и *getArea()* класса *Shape* имеют две разные реализации: реализацию в классе Square и другую – в классе Circle. У этих операций «много форм», следовательно, они полиморфны.

Рисунок 10.7 прекрасно иллюстрирует полиморфизм. Определен класс *Shape*, имеющий абстрактные операции *draw()* и *getArea()*.

> Полиморфизм означает «много форм». Полиморфные операции имеют много реализаций.

Классы Square и Circle наследуются от *Shape* и предоставляют реализации полиморфных операций *Shape::draw()* и *Shape::getArea()*. Все конкретные подклассы *Shape должны* предоставлять конкретные операции draw() и getArea(), потому что в надклассе они являются абстрактными. Это значит, что в draw() и getArea() все подклассы Shape можно интерпретировать (treat) одинаково. Таким образом, набор абстрактных операций является средством определения набора операций, которые *должны* быть реализованы *всеми* конкретными подклассами. Это называют контрактом.

> Конкретный подкласс должен реализовывать абстрактные операции, которые он наследует.

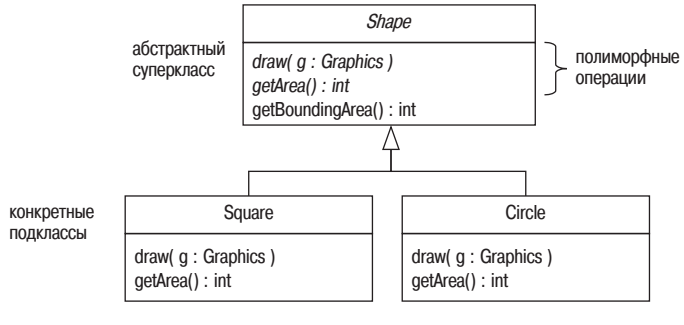


Рис. 10.7. *Иллюстрация полиморфизма*

Очевидно, что реализация операций draw() и getArea() для классов Square и Circle будет разной. Операция draw() для объектов класса Square будет отрисовывать квадраты, а для объектов класса Circle – круги. И реализации операции getArea() тоже будут разными. Для квадрата она будет возвращать width*height, а для круга – $\pi*r^2$. В этом суть полиморфизма: объекты разных классов имеют операции с *одинаковой* сигнатурой, но *разными* реализациями.

Инкапсуляция, наследование и полиморфизм – это «три столпа» ОО. Полиморфизм позволяет разрабатывать более простые системы, которые проще изменять, потому что разные объекты в них интерпретируются одинаково.

По сути, полиморфизм является важнейшим аспектом ОО, поскольку предоставляет возможность отправлять объектам *разных* классов *одинаковое* сообщение и получать от них соответствующий ответ. Иными словами, если послать объектам класса Square сообщение draw(), они отрисуют квадрат, а если послать такое же сообщение объектам класса Circle, они отрисуют круг. Объекты кажутся разумными.

10.4.1. Пример полиморфизма

Здесь представлен пример полиморфизма в действии. Предположим, есть класс Canvas (холст), обслуживающий коллекцию классов *Shape*. Хотя это и несколько упрощенная картина, работа многих графических систем действительно подобна этому. Модель такой простой графической системы приведена на рис. 10.8.

Нам уже известно, что создать экземпляр *Shape* (поскольку это абстрактный класс) нельзя. Но согласно принципу замещаемости можно создать экземпляры его конкретных подклассов и использовать их везде, где требуется класс *Shape*.

Итак, хотя на рис. 10.8 показано, что объекты типа Canvas содержат коллекцию объектов *Shape*, на самом деле в коллекцию могут входить только экземпляры конкретных подклассов *Shape*, потому что сам *Shape* является абстрактным и не может иметь экземпляров. В данном

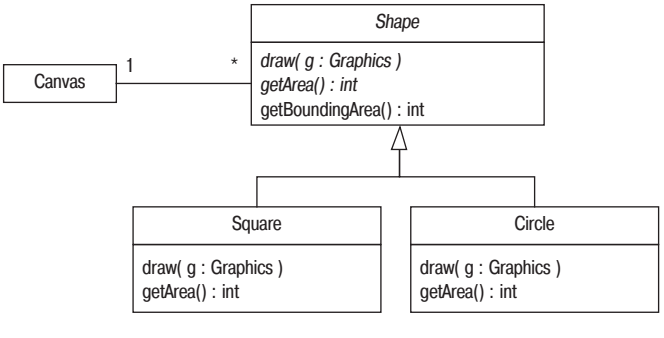


Рис. 10.8. *Диаграмма классов: полиморфизм в действии*

случае есть два конкретных подкласса, Circle и Square. Таким образом, коллекция может содержать объекты Circle и/или объекты Square.

> С полиморфизмом объекты разных классов по-разному отвечают на одно и то же сообщение.

На рис. 10.9 представлена модель объектов, соответствующая диаграмме классов, изображенной на рис. 10.8. Эта модель объектов показывает, что у объекта :Canvas имеется коллекция из четырех объектов *Shape*: s1, s2, s3 и s4, где s1, s3 и s4 – объекты класса Circle, а s2 – объект класса Square. Что происходит, когда объект :Canvas посылает каждому объекту этой коллекции сообщение draw()? Каждый объект, что и не удивительно, реагирует правильно: объекты Square отрисовывают квадраты, а объекты Circle – круги. Именно класс объекта определяет, что отрисовывает объект. Иначе говоря, класс объекта определяет семантику набора операций, предлагаемых объектом.

Главное здесь то, что каждый объект отвечает на сообщение вызовом соответствующей операции, заданной его классом. Все объекты одного класса будут отвечать на одно и то же сообщение вызовом одной и той же операции. Это не означает, что все объекты одного класса отвечают на одно сообщение совершенно одинаково. Результаты вызова операции обычно зависят от состояния объекта: значений всех его атрибутов и состояния всех отношений. Например, есть три объекта класса Square с разными значениями атрибутов width (ширина) и height (высота). Получив сообщение draw(), каждый из этих классов отрисует квадрат (т. е. значение или семантика операции остается неизменной), но все квадраты будут разного размера в зависимости от значений атрибутов width и height.

Вот еще один пример. Бизнес-правила снятия денег и вычисления процентов отличаются в зависимости от типа банковского счета. Так, для

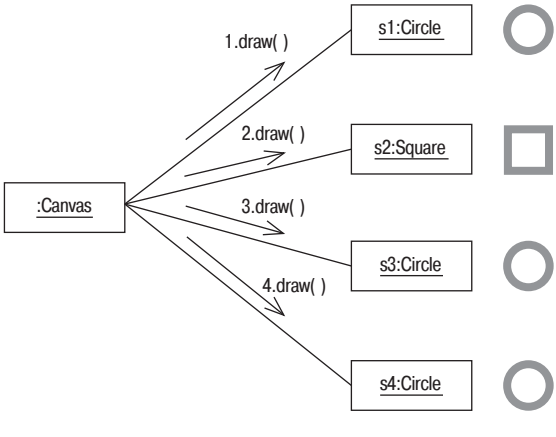


Рис. 10.9. *Объект Canvas имеет коллекцию из 4-х объектов Shape*

текущих счетов обычно существуют ограничения по превышению кредита, и поэтому они могут иметь отрицательный баланс, в то время как баланс депозитных счетов не может опускаться ниже нуля. Аналогично часто по-разному вычисляются и начисляются на счет проценты. Один из простых способов моделирования этих банковских операций показан на рис. 10.10. Описывается абстрактный класс *Account* (счет). Затем предоставляются конкретные подклассы CheckingAccount (текущий счет) и DepositAccount (депозитный счет). Абстрактный класс определяет абстрактные операции *withdraw()* (снять деньги) и *calculateInterest()* (вычислить процент), которые реализуются по-разному каждым из конкретных подклассов.

> Конкретные операции тоже могут быть полиморфными, но это считается плохим стилем.

Обратите внимание, что мы также переопределили конкретную операцию deposit() (разместить), предоставив ей новую реализацию в классе ShareAccount (паевой счет). Запомните, что для переопределения операции базового класса необходимо только предоставить подкласс с операцией, имеющей абсолютно аналогичную сигнатуру. Мы сделали это для ShareAccount, потому что бизнес-правила размещения денег на паевом счете (ShareAccount) отличаются от правил для других типов счетов (*Account*). Например, могут быть правила, определяющие минимальный размер вклада. Тогда будет две реализации deposit(): одна в *Account*, а другая в ShareAccount. Это значит, что теперь deposit() является полиморфной операцией. То есть конкретные операции, такие как deposit(), могут быть полиморфными!

Необходимо быть очень осторожным при переопределении конкретных операций, потому что при этом происходит не просто предоставление реализации операции абстрактного суперкласса – в этом случае изменяется существующая реализация. Выяснить допустимость этого можно, проверив спецификацию операции суперкласса и убедившись,

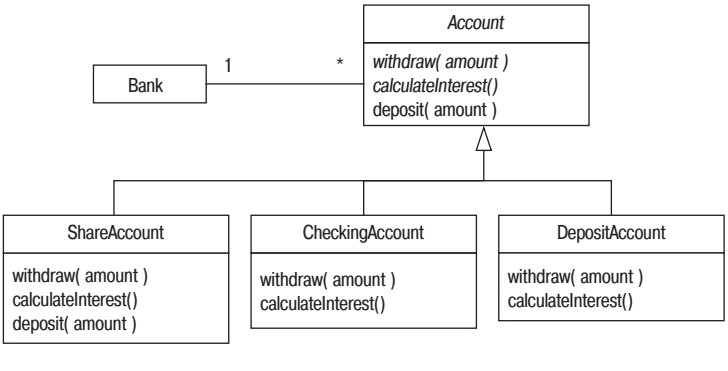


Рис. 10.10. *Диаграмма классов для банковских операций*

что при этом ее контракт не будет нарушен. Абстрактные операции можно безопасно переопределять всегда, потому что они именно для этого и предназначены. Однако переопределение конкретных операций может иметь неожиданные последствия и представлять некоторую опасность. Часто операция подкласса просто выполняет какое-то дополнительное действие и вызывает операцию суперкласса. Иначе говоря, она добавляет собственное поведение в операцию суперкласса. Это хороший способ повторного использования и расширения поведения конкретной операции суперкласса, поскольку, как правило, он безопасен.

Некоторые языки программирования обеспечивают возможность предотвратить переопределение конкретной операции суперкласса в подклассе. В Java добавление ключевого слова final (финальный) в сигнатуру операции явно запрещает переопределение операции. На практике в Java считается хорошим стилем определять как final все операции, кроме тех, которые вы хотите явно определить как полиморфные.

10.5. Дополнительные аспекты обобщения

В данном разделе мы рассмотрим два углубленных аспекта обобщения: множества обобщения и множества всех типов. Понятие множеств обобщения может быть весьма полезным. А вот множества всех типов используются крайне редко. Они включены в эту книгу главным образом для обеспечения полноты изложения.

10.5.1. Множества обобщения

Подклассы любого суперкласса можно организовать в одно или более множеств обобщения.

> Множества обобщения распределяют подклассы соответственно определенному правилу.

Множества обобщения группируют подклассы соответственно определенному правилу или на основании специализации. Обсудим пример. На рис. 10.11 показано, что у класса *Shape* может быть множество подклассов. Рассматривая эти подклассы, мы обнаруживаем здесь две совершенно разные группы фигур: двух- и трехмерные.

На диаграмме классов это разделение подклассов можно отобразить путем ассоциирования каждой группы фигур с разным множеством обобщения, как показано на рис. 10.12.

К множествам обобщения могут применяться ограничения. Они определяют, являются ли множества:

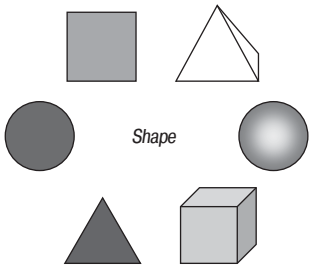


Рис. 10.11. *Класс Shape имеет множество подклассов*

- {complete} (полный) – подклассы множества обобщения охватывают *все* возможные варианты. Например, множество обобщения gender (пол) класса Person (человек), содержащего два подкласса, Male (мужчина) и Female (женщина), можно было бы считать {complete}, поскольку существует только два пола;
- {incomplete} (неполный) – могут существовать подклассы, кроме тех, которые представлены в множестве обобщения. Множество обобщения twoDShape явно {incomplete}, поскольку двухмерных фигур очень много;
- {disjoint} (несовместный) – объект может быть экземпляром *одного и только одного* из членов набора множества обобщения. Это самый распространенный случай;
- {overlapping} (перекрывающийся) – объект может быть экземпляром *нескольких* членов множества обобщения. Это довольно редкий случай, поскольку подразумевает множественное наследование или множественную классификацию.

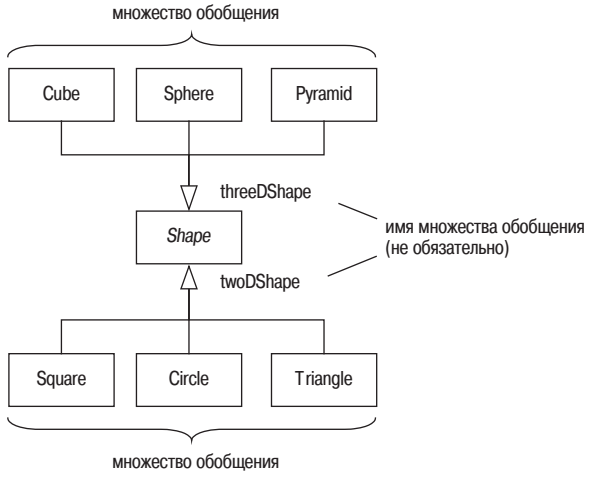


Рис. 10.12. *Множества обобщения на диаграмме классов*

Ограничения множеств обобщения могут комбинироваться, как показано в табл. 10.1.

Таблица 10.1

Ограничение	Множество полно	Члены множества могут иметь общие экземпляры
{incomplete, disjoint} – применяется по умолчанию	Нет	Нет
{complete, disjoint}	Да	Нет
{incomplete, overlapping}	Нет	Да
{complete, overlapping}	Да	Да

Рисунок 10.13 иллюстрирует применение ограничений множеств обобщения к примеру с классом *Shape*.

Как видите, множества обобщения – это аналитическая концепция, которая позволяет разделять множества подклассов на группы. Когда дело доходит до реализации, ни один из широко используемых ОО языков программирования не поддерживает напрямую множества обобщения, и с точки зрения реализации эта концепция является избыточной.

При реализации множества обобщения или игнорируются, или в качестве решения вводится новый уровень иерархии наследования, если в этом есть какой-то смысл. Взглянув на аналитическую модель на рис. 10.13, можно предположить, что существуют какие-то атрибуты или операции, общие для всех twoDShape и всех threeDShape. Это дает основание для перевода множеств обобщения в новые классы иерархии наследования, как показано на рис. 10.14.

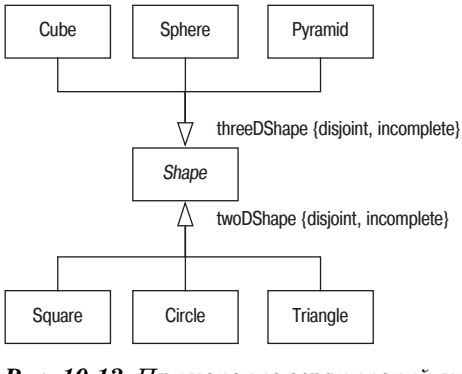


Рис. 10.13. *Применение ограничений множеств обобщения*

10.5.2. Множества всех типов

Множества всех типов (powertype) – это аналитическая концепция, которая в обычном повседневном моделировании встречается крайне

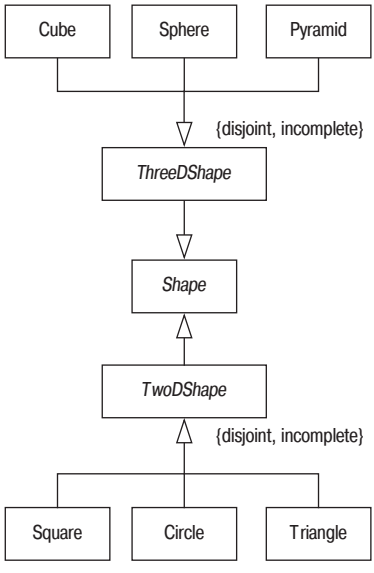


Рис. 10.14. *Множества обобщения переведены в новые классы иерархии наследования*

редко. Этот раздел включен в книгу в основном для обеспечения полноты изложения и как справочный материал, если вдруг кто-то из читателей когда-нибудь столкнется с этим понятием.

> Множество всех типов – это класс, экземпляры которого – классы, также являющиеся подклассами другого класса.

Множество всех типов – это класс, экземпляры которого являются классами. Эти экземпляры *также* являются подклассами другого класса.

Любой класс, экземпляры которого являются классами, называют *метаклассом* (класс класса). Таким образом, множество всех типов – это особый тип метакласса, экземпляры которого являются еще и подклассами другого класса.

Идея множеств всех типов довольно сложна. Лучше всего проиллюстрировать ее простым примером (рис. 10.15).

Во-первых, необходимо отметить, что InterestAccount (счет процентов) не является обычным классом; это множество всех типов, как обозначено стереотипом. Во-вторых, ассоциация между классами InterestAccount и *Account* не имеет обычной семантики ассоциации. В данном случае она показывает, что InterestAccount может быть (но необязательно (0..1)) множеством всех типов класса *Account* (и его подклассов благодаря наследованию).

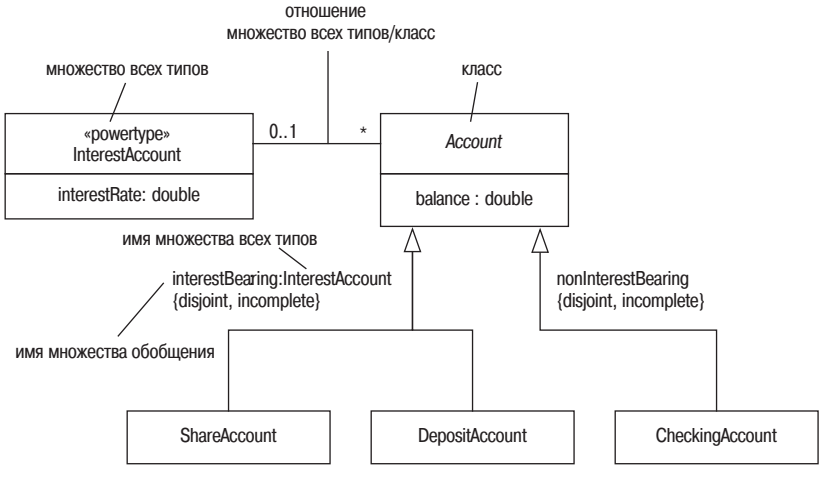


Рис. 10.15. *Пример применения множества всех типов*

Чтобы использовать множество всех типов, подклассы разделяются на один или более множеств обобщения. К одному или более этих множеств применяется множество всех типов. Тогда все классы такого набора обобщения становятся экземплярами этого множества всех типов.

Множество всех типов применяется к множеству обобщения путем указания имени множества всех типов после имени множества обобщения и двоеточия, так же как указывался бы тип атрибута после имени атрибута. Множество всех типов можно рассматривать как еще один тип для членов множества обобщения в дополнение к тому, который они получают от своего суперкласса.

На рис. 10.15 подклассы *Account* были разделены на два множества обобщения: interestBearing (приносящие проценты) и nonInterestBearing (беспроцентные), т. е. представляющие и не представляющие интереса. Множество обобщения interestBearing типизировано множеством всех типов InterestAccount. Это означает, что ShareAccount и DepositAccount одновременно являются подклассами *Account* и экземплярами InterestAccount. Они наследуют атрибут balance от класса *Account* и получают атрибут interestRate (процентная ставка) на основании того, что являются экземплярами InterestAccount.

Множество обобщения nonInterestBearing содержит единственный класс CheckingAccount – простой подкласс *Account*. Таким образом, CheckingAccount наследует атрибут balance от класса *Account*, но ничего не получает от InterestAccount.

Ни один из основных ОО языков программирования не поддерживает множества всех типов. Как же тогда можно применить это понятие на практике? На рис. 10.16 показано простое решение этой проблемы, где она реализуется посредством делегирования. В этом примере для

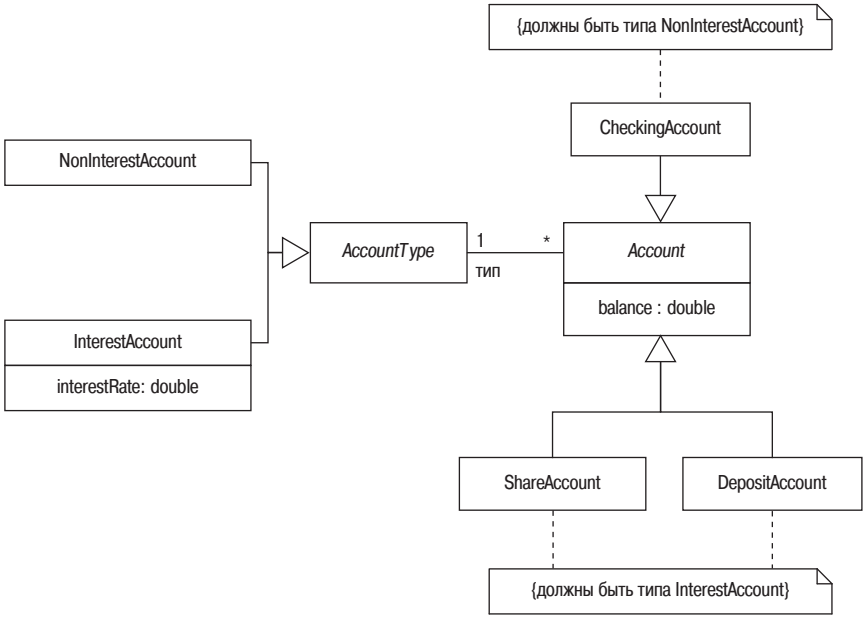


***Рис. 10.16**. Реализация множества всех типов посредством делегирования*

создания правильной иерархии наследования *AccountType* введены новые классы, *AccountType* и NonInterestAccount (беспроцентный счет). Типы каждого *Account* обозначены с помощью ограничений. Это довольно стандартный способ работы с множествами всех типов.

Теоретически множества всех типов предоставляют лаконичную и удобную идиому моделирования для аналитических моделей. Однако на практике они не используются или не понятны широкому кругу разработчиков. Таким образом, в случае применения они могут привести к полному замешательству. Множества всех типов не вносят ничего нового в набор моделирования и, возможно, никогда не будут даже поддерживаться средствами моделирования. Наш совет – избегайте их применения.

10.6. Что мы узнали

В этой главе были рассмотрены наследование и полиморфизм классов. Мы узнали следующее:

- Обобщение – это отношение между более общей и более специальной сущностями:
  - более специальная сущность полностью совместима с более общей сущностью;

    - принцип замещаемости гласит, что более специальная сущность может использоваться везде, где предполагается более общая сущность;
    - обобщение применяется ко всем классификаторам и некоторым другим элементам моделирования;
    - иерархии обобщения могут создаваться путем обобщения специальных сущностей или путем специализации общих сущностей;
    - все сущности, находящиеся на одном уровне иерархии обобщения, должны находиться на одном уровне абстракции.
- Наследование классов имеет место в отношении обобщения между классами.
    - Подкласс наследует от своих родителей атрибуты, операции, отношения и ограничения.
    - Подклассы могут:
        - добавлять новые возможности;
        - переопределять унаследованные операции:
            - подкласс предоставляет новую операцию с той же сигнатурой, что и родительская операция, которую он хочет переопределить;
            - сигнатура операции состоит из имени операции, типов всех параметров в заданном порядке и возвращаемого типа.
    - Абстрактные операции разработаны, чтобы не иметь реализации:
        - они выполняют роль структурного нуля;
        - все конкретные подклассы должны реализовывать все унаследованные абстрактные операции.
    - Абстрактный класс имеет одну или более абстрактных операций:
        - экземпляр абстрактного класса не может быть создан;
        - абстрактные классы определяют контракт как набор абстрактных операций, которые должны быть реализованы конкретными подклассами.
    - Полиморфизм означает «много форм». Он позволяет проектировать системы с абстрактными классами, которые во время выполнения замещаются конкретными подклассами – такие системы очень гибкие и легко расширяемые; просто вводятся дополнительные подклассы.
    - Полиморфные операции имеют несколько реализаций:
        - разные классы могут реализовывать одну и ту же полиморфную операцию по-разному;
        - полиморфизм позволяет экземплярам разных классов отвечать на одно и то же сообщение по-разному.
- Множество обобщения – набор подклассов, организованных по определенному правилу.

    - Ограничения:
        - {complete} – в множество обобщения входят все возможные члены;
        - {incomplete} – в множество обобщения входят не все возможные члены;
        - {disjoint} – объект может быть экземпляром не более чем одного из членов множества обобщения;
        - {overlapping} – объект может быть экземпляром нескольких членов множества обобщения;
        - {incomplete, disjoint} – применяется по умолчанию.
- Множество всех типов – класс, экземпляры которого – классы, являющиеся также подклассами другого класса.
    - Множество всех типов – это метакласс, экземпляры которого являются подклассами другого класса:
        - «powertype» – показывает, что класс является множеством всех типов.
    - Ассоциация между классом и множеством всех типов показывает, что класс может быть экземпляром множества всех типов.
    - Чтобы использовать множества всех типов:
        - подклассы разделяются на один или более множеств обобщения;
        - множество всех типов применяется к какому-либо множеству обобщения.

11

Пакеты анализа

11.1. План главы

В этой главе рассматриваются пакеты – механизм группировки UML – и их использование в анализе.

11.2. Что такое пакет?

Если вы помните основные принципы UML (раздел 1.8), то знаете, что к строительным блокам UML относятся сущности, отношения и диаграммы. Пакет – это группирующая сущность. Это контейнер и владелец элементов модели. У каждого пакета есть свое пространство имен, в рамках которого все имена должны быть уникальными.

> Пакет – это UML-механизм группировки сущностей.

По сути, пакет – это универсальный механизм организации элементов модели (включая другие пакеты) и диаграмм в группы. Он может использоваться для следующих целей:

- предоставления инкапсулированного пространства имен, в рамках которого все имена должны быть уникальными;
- группировки семантически взаимосвязанных элементов;
- определения «семантической границы» модели;
- предоставления элементов для параллельной работы и управления конфигурацией.

Пакеты позволяют создавать допускающую навигацию хорошо структурированную модель, обеспечивая возможность группировать сущности, имеющие близкие семантические связи. В модели можно устанавливать семантические границы, в пределах которых разные пакеты описывают разные аспекты функциональности системы.

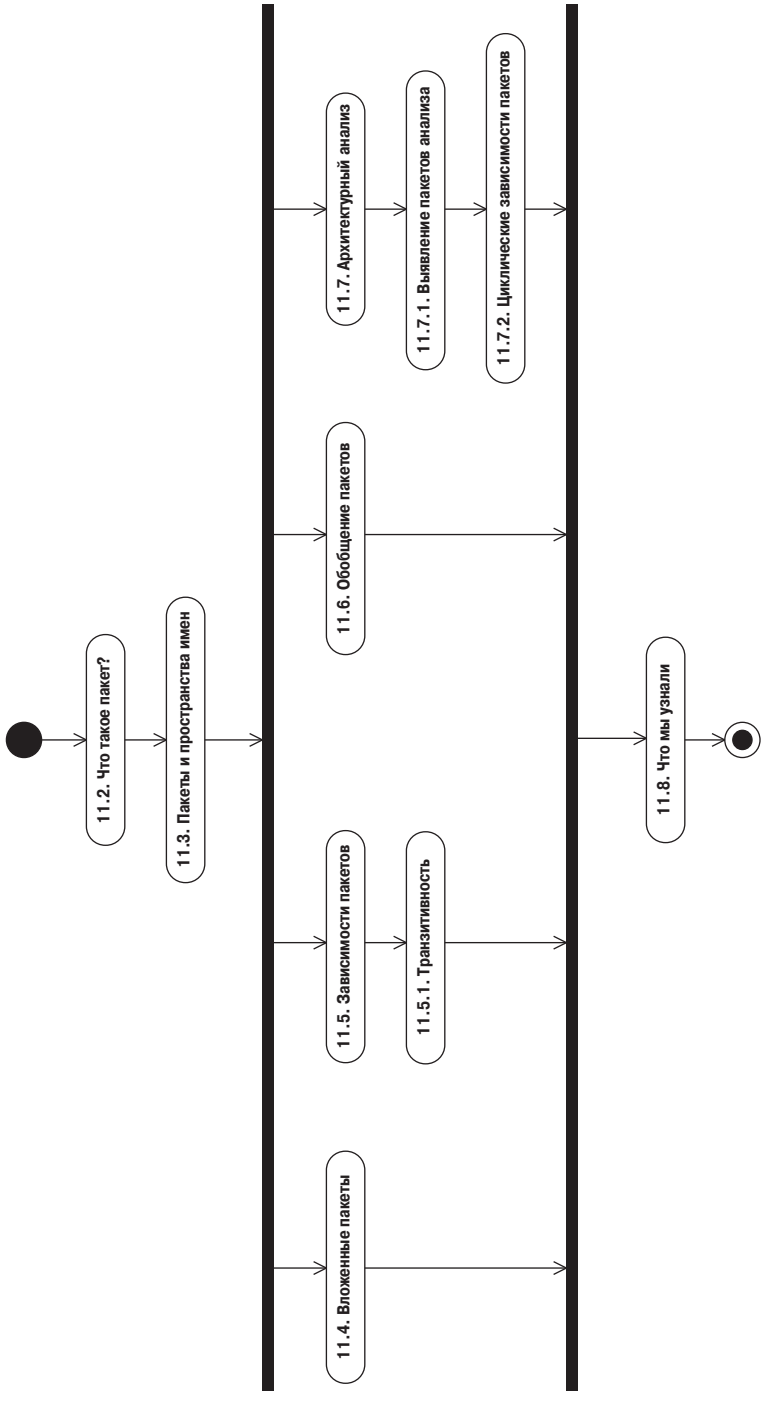


Рис. 11.1. План главы

Важно отметить, что в UML 2 пакет – это механизм *логической* группировки, предоставляющий пространство имен для своих членов. Если требуется физически сгруппировать элементы модели, должен использоваться компонент, который будет рассмотрен в разделе 22.2.

> Каждый элемент модели принадлежит одному пакету. Пакеты образуют иерархию.

Каждый элемент модели принадлежит только одному пакету. Иерархия принадлежности образует дерево, корнем которого является пакет высшего уровня. Для обозначения этого пакета может использоваться специальный стереотип UML «topLevel» (высший уровень). Если элемент моделирования явно не помещен в какой-либо пакет, он по умолчанию отправляется в пакет высшего уровня. Иерархия пакетов также образует иерархию пространства имен, в которой пакет высшего уровня является корнем пространства имен.

Пакеты анализа должны содержать:

- прецеденты;
- классы анализа;
- реализации прецедентов.

Синтаксис пакета UML довольно прост. Пиктограмма пакета – папка; имя пакета может быть указано на закладке, если содержимое пакета показано, или на теле папки. Синтаксис пакета показан на рис. 11.2. Здесь приведены три разных способа представления одного пакета на разных уровнях детализации.

Для элементов, находящихся в пакете, может быть задана видимость, показывающая, видят ли их клиенты пакета. Возможные значения видимости приведены в табл. 11.1.

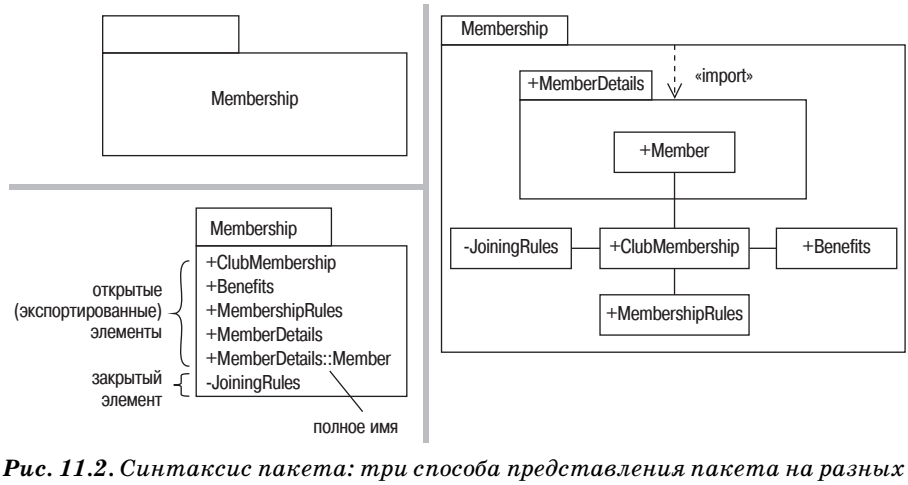


***Рис. 11.2.** Синтаксис пакета: три способа представления пакета на разных уровнях детализации*

Таблица 11.1

Символ	Видимость	Семантика
+	открытая (public)	Элементы с открытой видимостью видимы вне пакета – они *экспортируются* пакетом.
–	закрытая (private)	Элементы с закрытой видимостью полностью скрыты внутри пакета.

Видимость определяет, является ли элемент пакета видимым вне пакета.

Видимость элементов пакета может использоваться для управления количеством взаимосвязей между пакетами. Это возможно, потому что экспортируемые элементы пакета действуют как интерфейс, или окно в пакет. Этот интерфейс должен быть как можно меньше и проще.

Чтобы пакет гарантированно имел небольшой и простой интерфейс, необходимо свести до минимума количество открытых элементов пакета и максимально увеличить количество закрытых элементов. Реализовать это на этапе анализа может быть трудно, если не применить к ассоциациям возможность навигации. В противном случае между классами будет много двунаправленных ассоциаций, а классы, участвующие в ассоциации, должны или находиться в одном пакете, или оба быть открытыми. При проектировании отношения между классами становятся однонаправленными, и тогда открытым должен быть только класс-поставщик.

Для адаптации семантики пакетов под конкретные цели UML предоставляет два стандартных стереотипа (табл. 11.2).

Таблица 11.2

Стереотип	Семантика
«framework» (каркас)	Пакет, содержащий элементы модели, которые определяют многократно используемую архитектуру.
«modelLibrary» (библиотека модели)	Пакет, содержащий элементы, которые предназначены для повторного использования другими пакетами.

11.3. Пакеты и пространства имен

Пакет определяет так называемое инкапсулированное пространство имен. Это означает, что пакет создает границу, в рамках которой имена всех элементов должны быть уникальными. Это также означает, что если элементу из одного пространства имен необходимо обратиться к элементу из другого пространства имен, он должен указать и имя необходимого элемента, *и* путь к этому элементу, чтобы его можно было найти в пространствах имен. Этот путь навигации называют *полным именем* (*qualified name*) или *составным именем* (*pathname*) элемента.

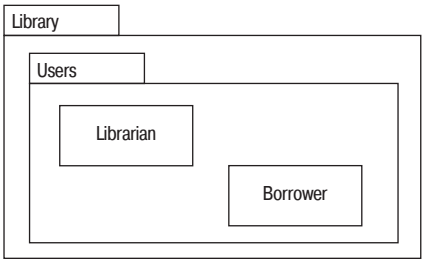


Рис. 11.3. *Синтаксис вложения путем встраивания*

Полное имя образуется именем элемента и именами пакетов, в которых он находится, перечисленными через двойные двоеточия. Первым указывается имя самого внешнего пакета, далее перечисляются все остальные пакеты в порядке вложенности вплоть до самого элемента. Полные имена очень похожи на составные имена в структурах каталогов.

Например, полное имя класса Librarian (библиотекарь) с рис. 11.3 будет таким:

```
Library::Users::Librarian
```

11.4. Вложенные пакеты

Пакеты могут быть вложены в другие пакеты с любой глубиной вложенности. Однако обычно достаточно всего двух или трех уровней. В противном случае модель может стать трудной для понимания и в ней будет сложно ориентироваться.

UML предлагает два способа представления вложенности. Первый очень нагляден, поскольку в нем элементы модели показаны физически вложенными в пакет. Пример представлен на рис. 11.3.

Альтернативный синтаксис вложения показан на рис. 11.4. Он удобен, когда необходимо представить глубокое или сложное вложение, которое может быть непонятным в случае отображения с помощью встраивания.

Вложенные пакеты имеют доступ к пространству имен своего пакета-владельца. Таким образом, элементы пакета Users (пользователи) на рис. 11.4 могут организовывать доступ ко всем элементам пакета Library (библиотека) через *не*полные имена. Однако обратное невозможно. Пакет-владелец для доступа к содержимому пакетов, которыми он владеет, должен использовать полные имена. Таким образом, в рассматриваемом примере элементы пакета Library для доступа к двум элементам пакета Users должны использовать полные имена: Users::Librarian и Users::Borrower.

В следующем разделе будет показано, как можно использовать зависимости для объединения пространств имен пакетов.

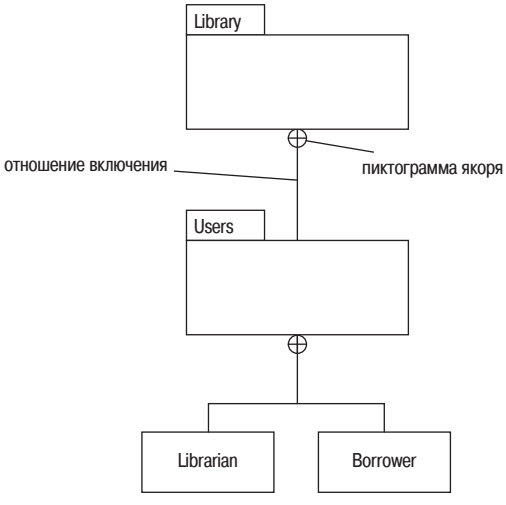


Рис. 11.4. *Синтаксис сложного вложения*

11.5. Зависимости пакетов

Между пакетами может быть установлено отношение зависимости.

> Отношение зависимости показывает, что один пакет некоторым образом зависит от другого.

Рассмотрим пример на рис. 11.5. Любой пакет, имеющий отношение зависимости с пакетом Membership (членство), сможет видеть открытые

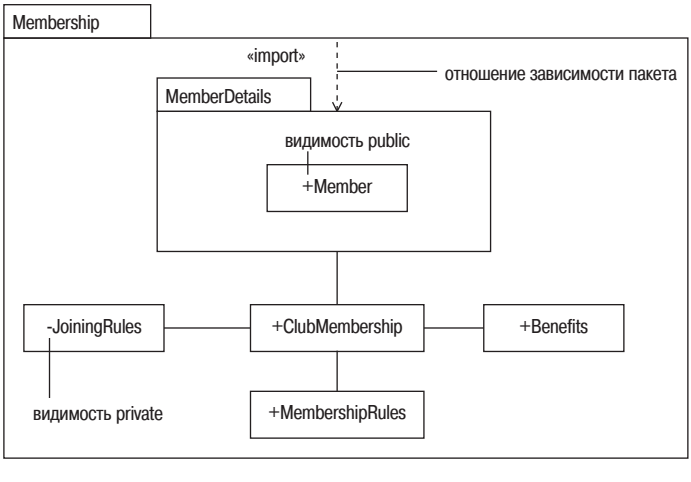
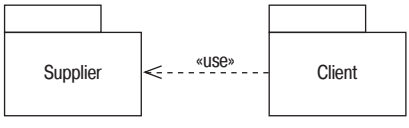
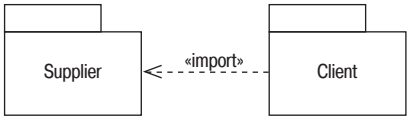
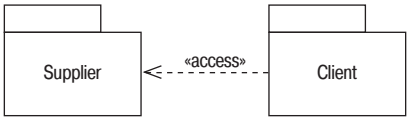
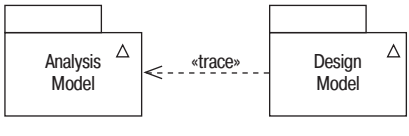
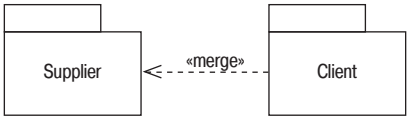


Рис. 11.5. *Структура пакета Membership*

элементы этого пакета (ClubMembership (членство в клубе), Benefits (льготы) и т. д.), но *не* будет видеть закрытый элемент JoiningRules (правила вступления).

Существует пять разных типов зависимостей между пакетами, каждый из которых имеет собственную семантику. Их обзор представлен в табл. 11.3.

Таблица 11.3

Отношение зависимости пакетов	Семантика
	Элемент клиентского пакета некоторым образом использует открытый элемент пакета-поставщика – клиент зависит от поставщика. Если зависимость пакета показана без стереотипа, необходимо предполагать зависимость «use».
	Открытые элементы пространства имен поставщика добавляются как открытые элементы в пространство имен клиента. Элементы клиента могут организовывать доступ ко всем открытым элементам поставщика через неполные имена.
	Открытые элементы пространства имен поставщика добавляются как закрытые элементы в пространство имен клиента Элементы клиента могут организовывать доступ ко всем открытым элементам поставщика с помощью неполных имен.
	«trace», как правило, представляет историческое развитие одного элемента в другую более развитую версию; обычно это отношение между моделями, а не элементами (межмодельное отношение).
	Открытые элементы пакета-поставщика объединяются с элементами клиентского пакета. Эта зависимость используется только при создании метамодели; в обычном ОО анализе и проектировании она не должна встречаться.

Зависимость «use» означает, что зависимости установлены скорее между *элементами* пакетов, а не между самими пакетами.

Зависимости «import» и «access» объединяют пространства имен клиента и поставщика. Это позволяет элементам клиента организовывать

доступ к элементам поставщика с помощью неполных имен. Разница между этими двумя зависимостями в том, что «import» осуществляет открытое объединение, т. е. объединенные элементы поставщика становятся открытыми в клиенте. Тогда как «access» проводит закрытое объединение, т. е. объединенные элементы становятся в клиенте закрытыми.

«trace» – «останется только один». Тогда как другие зависимости пакетов устанавливаются между сущностями одной модели, «trace» обычно представляет некоторое историческое развитие одного пакета в другой. Поэтому «trace» часто отражает отношения между *разными* моделями. Полная UML-модель может быть представлена в виде пакета с маленьким треугольником в верхнем правом углу. В табл. 11.3 показана межмодельная зависимость «trace» между аналитической моделью и проектной моделью. Очевидно, что такая диаграмма является метамоделью, в которой моделируются отношения между моделями! Как таковая она используется не очень часто.

«merge» – сложное отношение, которое обозначает ряд трансформаций между элементами пакета-поставщика и пакета-клиента. Элементы пакета-поставщика объединяются с элементами клиента для создания новых, расширенных клиентских элементов. Эта зависимость используется только при создании метамоделей (например, она широко применяется в метамодели UML) и *не* должна применяться в обычном ОО анализе и проектировании. Далее в книге она нигде не обсуждается. Дополнительную информацию можно найти в книге [Rumbaugh 1].

11.5.1. Транзитивность

Транзитивность (transitivity) – термин, применяемый к отношениям. Он означает, что если существует отношение между сущностями A и B и отношение между сущностями B и C, то существует неявное отношение между сущностями A и C.

> **«import»** – транзитивное отношение, **«access»** – нет.

Важно отметить, что зависимость «import» – транзитивная, а зависимость «access» – нет. Как было показано выше, это объясняется тем, что при наличии зависимости «import» между клиентом и поставщиком открытые элементы пакета-поставщика становятся открытыми элементами клиента. Эти импортированные открытые элементы доступны вне клиентского пакета. С другой стороны, когда между клиентом и поставщиком установлена зависимость «access», открытые элементы пакета-поставщика становятся закрытыми элементами клиента. Эти закрытые элементы *не доступны* вне клиентского пакета.

Рассмотрим пример, приведенный на рис. 11.6. Пакет A обеспечивает доступ к пакету B, а пакет B – к пакету C.

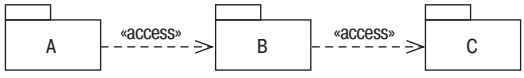


Рис. 11.6. *«access» – нетранзитивная зависимость*

Отсутствие транзитивности в «access» означает следующее:

- открытые элементы пакета C становятся закрытыми элементами пакета B;
- открытые элементы пакета B становятся закрытыми элементами пакета A;
- следовательно, элементы пакета A *не имеют* доступа к элементам пакета C.

Эта нетранзитивность «access» позволяет активно управлять и контролировать внутреннюю связность и целостность модели. Доступно только то, что указано *явно*.

11.6. Обобщение пакетов

Обобщение пакетов во многом подобно обобщению классов. В обобщении пакетов более специализированные дочерние пакеты наследуют открытые элементы родительского пакета. Дочерние пакеты могут вводить новые элементы и переопределять элементы родительского пакета, предоставляя альтернативный элемент с тем же именем.

В примере на рис. 11.7 пакеты Hotels (гостиницы) и CarHire (прокат автомобилей) наследуют все открытые элементы своего родительского

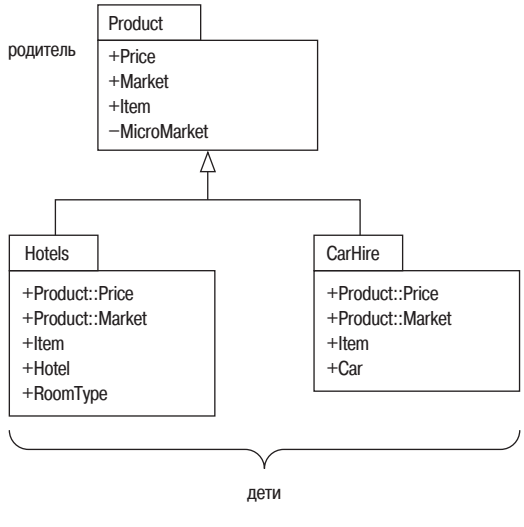


Рис. 11.7. *Дочерние элементы наследуют элементы родителя*

пакета Product (продукт). Но пакеты Hotels и CarHire переопределяют класс Item (элемент), унаследованный от родителя, предоставляя альтернативный класс с тем же именем. Дочерние пакеты также могут добавлять новые элементы. Пакет Hotels вводит классы Hotel и RoomType (тип комнаты), а пакет CarHire вводит класс Car (автомобиль).

> Дочерние пакеты наследуют элементы от своих родителей. Они могут переопределять родительские элементы. Они могут вводить новые элементы.

Как и при наследовании классов, должен применяться принцип замещаемости: везде, где может использоваться пакет Product, должна существовать возможность применять или Hotels, или CarHire.

11.7. Архитектурный анализ

В архитектурном анализе взаимосвязанные классы анализа объединяются в ряд пакетов анализа. Затем организуются разделы и уровни, как показано на рис. 11.8. Каждый пакет анализа на определенном уровне – это раздел.

> Архитектурный анализ группирует взаимосвязанные классы в пакеты анализа, а затем распределяет пакеты по уровням.

Одна из целей архитектурного анализа – попытаться минимизировать число взаимосвязей в системе. Сделать это можно тремя способами:

- сократить до минимума зависимости между пакетами анализа;
- сократить до минимума число открытых элементов в каждом пакете анализа;
- максимально увеличить число закрытых элементов в каждом пакете анализа.

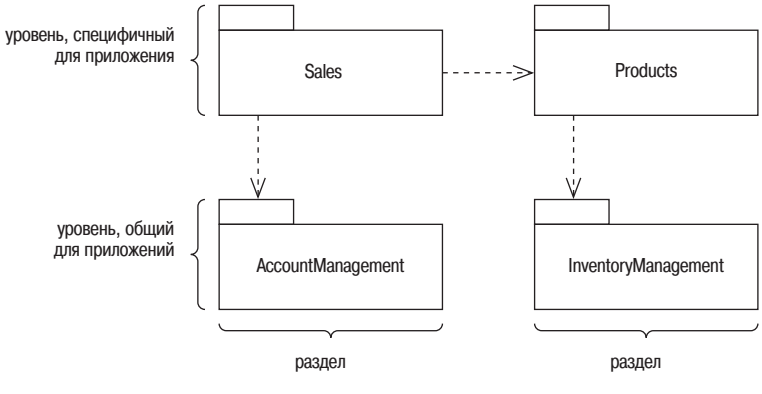


Рис. 11.8. *Пакеты распределяются по уровням*

> Всегда сокращайте до минимума количество взаимосвязей.

Сокращение количества взаимосвязей – одна из наиболее важных задач архитектурного анализа, потому что системы с высокой степенью связанности обычно сложны для построения и обслуживания. Всегда необходимо пытаться обеспечить минимально необходимую связанность.

По мере углубления и превращения модели в проектную количество уровней будет расти. Но на этапе анализа пакеты можно просто распределить по двум уровням: специфичному и общему для приложений. Специфичный уровень содержит функциональность, абсолютно специфичную для конкретного приложения. Общий уровень содержит более широко используемые функциональные возможности. На рис. 11.8 AccountManagement (ведение счетов) и InventoryManagement (управление материально-техническими ресурсами) могут использоваться в нескольких разных приложениях, поэтому эти пакеты, как и следовало ожидать, находятся в общем уровне.

11.7.1. Выявление пакетов анализа

> Ищите группы классов, образующие связный элемент.

Выявление пакетов анализа осуществляется путем идентификации групп элементов модели, имеющих прочные семантические связи. Пакеты анализа часто обнаруживаются со временем в ходе развития и совершенствования модели. Пакеты анализа обязательно должны отражать реальные семантические группы элементов, а не просто некоторое идеализированное (но выдуманное) представление логической архитектуры.

Где начинать поиск таких групп? Статическая модель – самый полезный источник пакетов. Необходимо искать следующее:

- связные группы классов на диаграммах классов;
- иерархии наследования.

В качестве источника пакетов можно также рассматривать модель прецедентов. Важно попытаться сделать пакеты максимально связанными с точки зрения бизнес-процесса. Обычно прецеденты *пересекают* несколько пакетов анализа: один прецедент может реализовываться классами из нескольких разных пакетов. Тем не менее один или более прецедентов, поддерживающих определенный бизнес-процесс, или актера, или ряд взаимосвязанных прецедентов, *могут* указывать на потенциальный пакет.

После идентификации ряда предполагаемых пакетов необходимо попытаться сократить до минимума количество открытых членов и зависимостей между пакетами. Это осуществляется путем:

- перемещения классов из пакета в пакет,
- добавления пакетов,
- удаления пакетов.

Основой хорошей структуры пакетов является высокая связность *в рамках* пакета и низкая связанность *между* пакетами. Пакет должен содержать группу тесно взаимосвязанных классов. Самую тесную взаимосвязь классов образует наследование (глава 10), далее композиция (глава 18), агрегирование (глава 18) и наконец зависимости (глава 9). Классы, образующие иерархии наследования или композиции, являются основными кандидатами на размещение в одном пакете. Это обеспечит высокую связность в рамках пакета и, вероятно, более низкую связанность с другими пакетами.

И как всегда, при создании аналитической модели пакетов необходимо придерживаться простоты. Важнее получить правильный набор пакетов, чем широко применять такие возможности, как обобщение пакетов и стереотипы зависимостей. Все это можно добавить позже и только в том случае, если это сделает модель более понятной. Отсутствие вложенных пакетов – один из гарантов простоты модели. Чем глубже что-то помещено в структуру вложенных пакетов, тем более непонятным оно становится. Нам встречались модели с очень глубоко вложенными пакетами, содержащими лишь один-два класса. Такие модели больше похожи на стандартную функциональную декомпозицию, чем на объектную модель.

В пакете должно быть от четырех до десяти классов анализа. Но могут быть и исключения из правил. Если нарушение этого правила делает модель более ясной, нарушайте его! Иногда в модель пакетов необходимо ввести пакеты с одним-двумя классами, чтобы избавиться от циклической зависимости. В таких случаях это вполне обоснованно.

На рис. 11.9 показан пример аналитической модели пакетов для простой системы электронной коммерции. Эта система представлена как учебный пример на нашем веб-сайте *www.umlandtheunifiedprocess.com*.

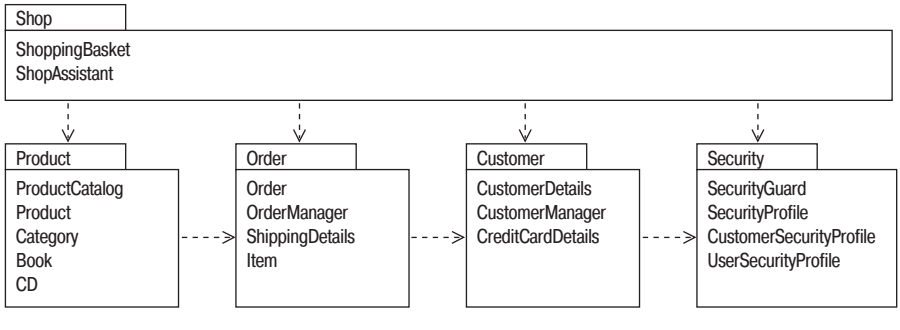


Рис. 11.9. *Модель пакетов анализа для системы е-коммерции*

11.7.2. Циклические зависимости пакетов

В аналитической модели пакетов необходимо избегать циклических зависимостей. Если пакет A некоторым образом зависит от пакета B, и наоборот, то это очень сильный аргумент в пользу объединения этих двух пакетов. Объединение пакетов (merging) является хорошим способом устранения циклических зависимостей. Но лучше, и это очень часто срабатывает, попытаться вынести общие элементы в третий пакет C и удалить цикл, перестроив отношения зависимостей. Это отображено на рис. 11.10.

> Избегайте циклических зависимостей между пакетами.

Многие инструментальные средства моделирования позволяют автоматически проверять зависимости между пакетами. Такой инструмент создает список нарушений прав доступа, если элемент одного пакета организовывает доступ к элементу другого пакета, но между этими двумя пакетами не установлена видимость или зависимость.

В аналитической модели порой невозможно создать диаграмму пакетов без нарушения прав доступа, поскольку в анализе часто используются двунаправленные отношения между классами. Предположим, имеется очень простая модель: один класс находится в пакете A, другой класс – в пакете B. Если класс пакета A имеет двунаправленное отношение с классом пакета B, тогда пакет A зависит от пакета B, но и пакет B зависит от пакета A. Имеем циклическую зависимость между двумя па-

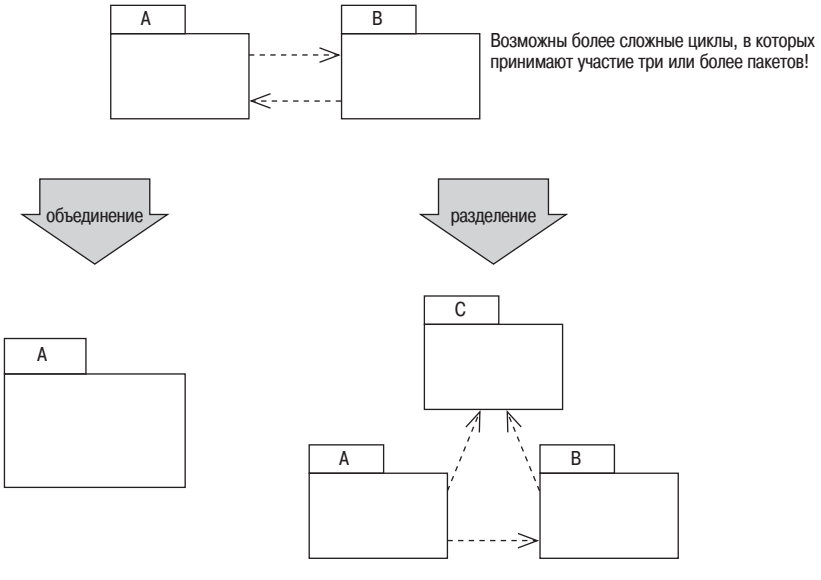


Рис. 11.10. *Два способа устранения циклических зависимостей*

кетами. Единственная возможность избавиться от этого нарушения – уточнить отношение между A и B, сделав его однонаправленным, либо поместить оба класса в один пакет. Таким образом, зависимости пакетов являются превосходным аргументом в пользу применения возможности навигации в аналитических моделях! С другой стороны, классы, действительно имеющие взаимные зависимости (а не зависимости, являющиеся результатом несовершенства модели), должны находиться в одном пакете.

11.8. Что мы узнали

В этой главе были рассмотрены пакеты анализа. В частности, было показано, как можно максимально увеличить связность пакета анализа и свести до минимума количество связей между пакетами анализа. Это помогает создавать более надежные и удобные в обслуживании системы. Мы узнали следующее:

- Пакет – это UML-механизм группировки сущностей.
- Пакеты служат многим целям:
  - группируют семантически взаимосвязанные элементы;
  - образуют «семантическую границу» в модели;
  - обеспечивают элементы управления конфигурацией;
  - в проектировании они обеспечивают элементы для распараллеливания работы;
  - предоставляют инкапсулированное пространство имен, в котором все имена должны быть уникальными – чтобы организовать доступ к элементу пространства имен, необходимо указать и имя элемента, и имя пространства имен.
- Каждый элемент модели принадлежит одному пакету:
  - пакеты образуют иерархию;
  - корневой пакет может быть обозначен стереотипом «topLevel»;
  - по умолчанию элементы модели помещаются в пакет «topLevel».
- В пакетах анализа могут содержаться:
  - прецеденты;
  - классы анализа;
  - реализации прецедентов.
- Элементы пакета могут иметь видимость:
  - видимость используется для управления количеством взаимосвязей между пакетами;
  - существует два уровня видимости:
    - открытый (+) – элементы видимы другим пакетам;
    - закрытый (–) – элементы полностью скрыты.

- Стереотипы пакетов:
  - «framework» – пакет, содержащий элементы модели, которые определяют многократно используемую архитектуру;
  - «modelLibrary» – пакет, содержащий элементы, предназначенные для использования другими пакетами.
- Пакет определяет инкапсулированное пространство имен:
  - для обращения к элементам других пакетов используются полные имена, например

    Library::Users::Librarian

- Вложенные пакеты:
  - внутренний пакет может видеть все открытые члены своих внешних пакетов;
  - если между внешним пакетом и членами его внутренних пакетов не установлена явная зависимость (обычно «access» или «import»), он не может видеть члены своих внутренних пакетов; это обеспечивает возможность скрывать детали реализации во вложенных пакетах.
- Отношение зависимости между пакетами указывает на то, что клиентский пакет некоторым образом зависит от пакета-поставщика.
  - «use» – элемент клиентского пакета использует открытый элемент пакета-поставщика.
  - «import» – открытые элементы пространства имен поставщика добавляются как открытые элементы в пространство имен клиента. Элементы клиента имеют доступ ко всем открытым элементам поставщика через неполные имена.
  - «access» – открытые элементы пространства имен поставщика добавляются как закрытые элементы в пространство имен клиента. Элементы клиента имеют доступ ко всем открытым элементам поставщика через неполные имена.
  - «trace» – клиент является историческим развитием поставщика. Это отношение обычно применяется к моделям, а не к элементам.
  - «merge» – открытые элементы пакета-поставщика объединяются с элементами клиентского пакета. Используется только при разработке метамоделей.
- Транзитивность: если у A есть отношение с B, и у B есть отношение с C, значит, A имеет отношение с C.
  - «import» транзитивно.
  - «access» нетранзитивно.
- Обобщение пакетов:
  - очень похоже на обобщение классов;

    - дочерние пакеты:
        - наследуют элементы от родительского пакета;
        - могут вводить новые элементы;
        - могут переопределять родительские элементы.
- Архитектурный анализ:
    - распределяет связные наборы классов анализа в пакеты анализа;
    - распределяет пакеты анализа по уровням согласно их семантике;
    - пытается минимизировать количество взаимосвязей путем:
        - сокращения до минимума зависимостей пакетов;
        - сокращения до минимума количества открытых элементов во всех пакетах;
        - доведения до максимума количества закрытых элементов во всех пакетах.
- Выявление пакетов анализа.
    - Рассматриваются классы анализа с целью поиска:
        - связных групп тесно взаимосвязанных классов;
        - иерархий наследования;
        - самую тесную взаимосвязь классов образуют (в порядке убывания) наследование, композиция, агрегирование, зависимость.
    - Рассматриваются прецеденты:
        - в группах прецедентов, поддерживающих определенный бизнес-процесс или актера, могут быть классы анализа, которые должны быть объединены в один пакет;
        - среди взаимосвязанных классов могут быть классы анализа, которые должны быть объединены в один пакет;
        - будьте внимательны: пакеты анализа часто охватывают несколько прецедентов!
    - Модель пакетов должна быть уточнена с целью максимального увеличения связности в рамках пакетов и сокращения до минимума зависимостей между пакетами. Это достигается путем:
        - перемещения классов из пакета в пакет;
        - добавления пакетов;
        - удаления пакетов;
        - удаления циклических зависимостей путем объединения пакетов или разделения их для выделения объединенных классов.

12

Реализация прецедентов

12.1. План главы

В этой главе обсуждается процесс реалG294

изации прецедентов, в которых осуществляется моделирование взаимодействий объектов.

12.2. Деятельность UP: Анализ прецедента

В предыдущих главах было показано, как создается класс анализа – артефакт деятельности Анализ прецедента. Второй артефакт, получаемый в результате этой деятельности, – реализация прецедента, как показано на рис. 12.2. Входные артефакты этой деятельности рассматривались в разделе 8.2.

Аналитическая модель классов – это статическая структура системы, а реализации прецедентов показывают, как взаимодействуют экземпляры классов анализа для осуществления функциональности системы. Это часть динамического представления системы.

Цели разработчика при реализации прецедентов во время фазы анализа следующие:

- Выяснить, взаимодействие каких классов анализа обеспечивает поведение, определенное прецедентом; во время реализации прецедентов могут быть обнаружены новые классы анализа.
- Выяснить, какими сообщениями должны обмениваться экземпляры этих классов для реализации заданного поведения. Как будет показано в этой главе, это выявит:
  - ключевые операции, необходимые классам анализа;
  - ключевые атрибуты классов анализа;
  - важные отношения между классами анализа.

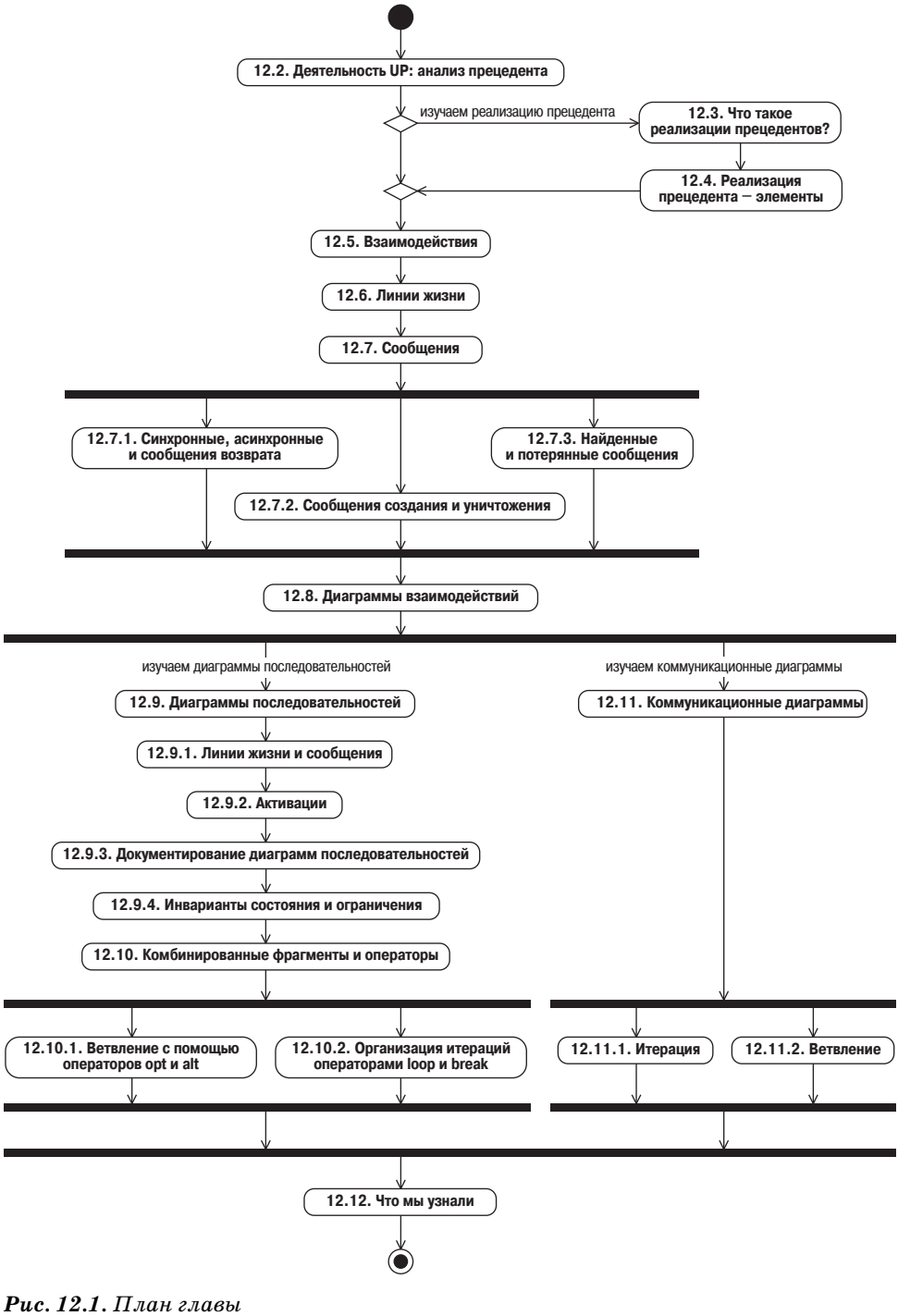


Рис. 12.1. *План главы*

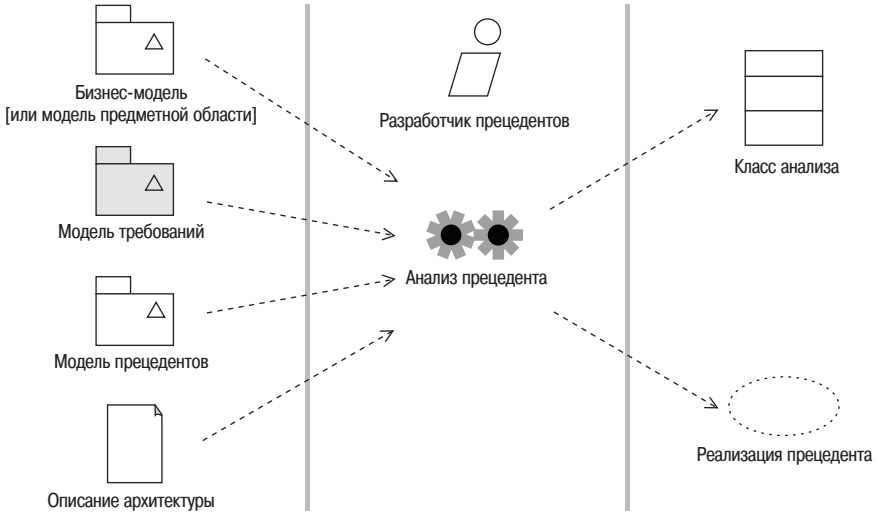


Рис. 12.2. *Два артефакта деятельности Анализ прецедента. Адаптировано с рис. 8.25 [Jacobson 1] с разрешения издательства Addison-Wesley*

- Скорректировать модель прецедентов, модель требований и классы анализа, добавив в них информацию, полученную при реализации прецедентов. Все модели должны быть согласованы и синхронизированы друг с другом.

При реализации прецедентов в процессе анализа важно сосредоточиться на отражении *ключевых* атрибутов, операций и отношений между классами анализа. На этом этапе не надо заниматьсятакими деталями, как параметры операций. Эта информация будет раскрыта при проектировании.

Также не надо реализовывать все прецеденты. Необходимо выбрать и проработать только самые основные. Реализация прецедентов должна продолжаться до тех пор, пока разработчик не почувствует, что обладает достаточной информацией для понимания совместной работы классов анализа. Получив необходимый набор информации, остановитесь. UP – итеративный процесс. Поэтому при необходимости вы сможете доработать реализацию прецедентов позже.

По завершении реализации прецедентов в процессе анализа вы получите аналитическую модель, обеспечивающую высокоуровневое представление динамического поведения системы.

12.3. Что такое реализации прецедентов?

После нахождения классов анализа ключевым моментом в анализе является поиск реализаций прецедентов. Они включают наборы классов, реализующих поведение, определенное в прецеденте. Пусть, к приме-

Рис. 12.3. *Синтаксис реализации прецедента*

ру, есть прецедент BorrowBook (взять книгу на время) и идентифицированы классы анализа Book (книга), Ticket (формуляр), Borrower (пользователь библиотеки) и актер Librarian (библиотекарь). Необходимо создать реализацию прецедента, демонстрирующую взаимодействие этих классов и их объектов, которое направлено на реализацию поведения, определенного в BorrowBook. Таким образом, прецедент, который является описанием функциональных требований, превращается в диаграммы классов и взаимодействий, а это уже высокоуровневое описание системы.

> Реализации прецедентов показывают, как взаимодействуют классы, чтобы реализовать функциональность системы.

Хотя UML предлагает символ для реализаций прецедентов (рис. 12.3), они редко представляются в моделях явно. Причина состоит в том, что у каждого прецедента *только одна* реализация. Таким образом, создание диаграммы реализаций прецедентов не привнесет никакой дополнительной информации. Соответствующие элементы (табл. 12.1) просто добавляются в инструмент моделирования, и реализации прецедентов становятся неявной частью базы (задним планом) модели.

> Реализации прецедентов – это неявная часть базы модели; обычно они не отображаются на диаграммах.

Таблица 12.1

Элемент	**Назначение**
Диаграммы классов анализа	Показывают классы анализа, взаимодействующие для реализации прецедента.
Диаграммы взаимодействий	Показывают взаимодействия определенных экземпляров, реализующих прецедент; это «моментальные снимки» работающей системы.
Особые требования	Процесс реализации прецедентов может выявить новые характерные для прецедента требования; они должны быть зафиксированы.
Уточнение прецедентов	Во время реализации может быть обнаружена новая информация, т. е. происходит обновление исходного прецедента.

12.4. Реализация прецедента – элементы

Реализации прецедентов включают элементы, показанные в табл. 12.1.

По сути, реализация прецедента – это процесс уточнения. Вы выбираете один из аспектов поведения системы, зафиксированный в прецеденте и любых ассоциированных с ним требованиях, и моделируете то, как это все может быть реализовано через взаимодействия экземпляров классов анализа, которые были найдены. Тем самым вы проходите путь от общей спецификации требуемого поведения до довольно подробного описания взаимодействий между классами и объектами, которые сделают это поведение реальным.

> Диаграммы классов анализа «рассказывают историю» об одном или более прецедентах.

Диаграммы классов анализа – жизненно важная часть реализации прецедента. Они должны «рассказывать истории» о системе: о том, как должны быть взаимосвязаны классы, чтобы их экземпляры могли взаимодействовать для реализации поведения, определенного в одном или более прецедентах.

> Диаграммы взаимодействий показывают, как взаимодействуют экземпляры классификаторов для реализации поведения системы.

Кроме диаграмм классов можно создать диаграммы, демонстрирующие совместную работу и взаимодействие экземпляров этих классов анализа, направленные на реализацию части или всего поведения прецедента. Эти диаграммы называют диаграммами взаимодействий. Существует четыре типа таких диаграмм: диаграммы последовательностей, коммуникационные диаграммы, диаграммы обзора взаимодействий и временные диаграммы. Диаграммы последовательностей и коммуникационные обсуждаются в этой главе, диаграммы обзора взаимодействий – в разделе 15.12, а временные диаграммы – в разделе 20.7.

ОО моделирование – итеративный процесс. Поэтому не надо удивляться, если при более глубоком моделировании будут выявлены новые требования или понадобится изменить существующие прецеденты. Все это – часть реализации прецедентов. Существующие документы должны соответствовать самой последней информации о системе. Необходимо обновлять модель прецедентов, модель требований и классы анализа, чтобы все они были согласованными.

12.5. Взаимодействия

Взаимодействия – это просто единицы поведения классификатора. Этот классификатор, который называют *контекстным классификатором* (*context classifier*), предоставляет контекст взаимодействия.

> Взаимодействие – это единица поведения контекстного классификатора.

Взаимодействие может использовать любую из возможностей своего контекстного классификатора или любые возможности, к которым классификатор имеет доступ (например, временные или глобальные переменные).

В реализации прецедента контекстным классификатором является прецедент. Создается одно или более взаимодействий, чтобы показать, как экземпляры классификаторов (в данном случае классов анализа), обмениваясь сообщениями, могут реализовать определенное прецедентом поведение.

В ходе работы над диаграммами взаимодействий обнаруживается все больше и больше операций и атрибутов классов анализа. Как часть процесса реализации прецедента должно осуществляться обновление диаграмм классов анализа этой информацией.

Ключевые элементы диаграмм взаимодействий – линии жизни и сообщения. Они подробно рассматриваются в следующих двух разделах.

12.6. Линии жизни

> Линия жизни – участник взаимодействия.

Линия жизни (*lifeline*) представляет одного участника взаимодействия, т. е. она представляет, как экземпляр конкретного классификатора участвует во взаимодействии. Синтаксис линии жизни приведен на рис. 12.4.

У каждой линии жизни есть необязательное имя, тип и необязательный селектор.

- Имя используется для обращения к линии жизни во взаимодействии.
- Тип – имя классификатора, экземпляр которого представляет линия жизни.
- Селектор – логическое условие, которое может использоваться для выбора единственного экземпляра, удовлетворяющего этому условию. Если селектора нет, линия жизни ссылается на произвольный экземпляр классификатора. Селекторы действительны, только ес-

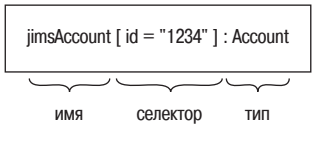


Рис. 12.4. *Синтаксис линии жизни*

ли кратность типа больше единицы, т. е. существует множество экземпляров, из которых можно выбирать. На рис. 12.4 селектор выбирает экземпляр класса Account, id которого – «1234».

Линии жизни изображаются той же пиктограммой, что и их тип, и имеют вертикальный пунктирный «хвост», когда используются в диаграммах последовательностей. Некоторые примеры линий жизни приведены на рис. 12.5.

> Линии жизни представляют, как экземпляр классификатора участвует во взаимодействии.

Линию жизни объекта можно рассматривать как элемент, который представляет, *как* экземпляр классификатора может участвовать во взаимодействии. Однако она не представляет никакого *конкретного* экземпляра классификатора. Это едва различимое, но важное отличие. Взаимодействие описывает, как экземпляры классификатора взаимодействуют в общем, а не выделяет какое-то одно конкретное взаимодействие между рядом конкретных экземпляров. Поэтому можно считать, что линия жизни объекта представляет *роль*, которую может играть во взаимодействии экземпляр классификатора.

Реальные экземпляры можно показать прямо на диаграмме взаимодействий. Используется обычная нотация для экземпляров: символ классификатора с именем экземпляра, селектор (если таковой имеется), двоеточие и имя классификатора; все подчеркивается.

Различие между линиями жизни и экземплярами послужило причиной появления двух разных форм диаграмм взаимодействий. *Общая форма* (*generic form*) диаграммы взаимодействий показывает взаимодействие между линиями жизни, представляющими произвольные экземпляры. *Форма экземпляров* (*instance form*) диаграммы взаимодействий показывает взаимодействие между конкретными экземплярами. Общая форма диаграммы является более удобной и часто используемой.

Чтобы взаимодействие было полным, должны быть определены сообщения, посылаемые между линиями жизни. Сообщения рассматриваются в следующем разделе.

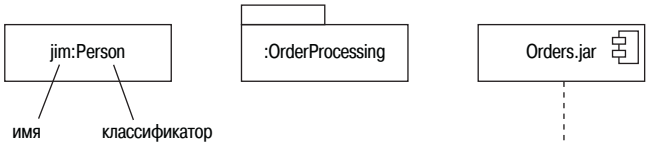


Рис. 12.5. *Примеры пиктограмм линии жизни*

12.7. Сообщения

> Сообщение – это особый вид коммуникации между линиями жизни.

Сообщение представляет особый тип коммуникации между двумя линиями жизни. Такое взаимодействие может включать:

- вызов операции – сообщение вызова;
- создание или уничтожение экземпляра – сообщение создания или уничтожения;
- отправку сигнала.

Сообщение вызова, получаемое линией жизни, является запросом на вызов операции, имеющей аналогичную сообщению сигнатуру. Таким образом, для каждого поступающего на линию жизни сообщения вызова в классификаторе этой линии жизни должна существовать соответствующая операция. UML допускает несовпадение (рассинхронизацию) сообщений и операций на диаграммах взаимодействий, что обеспечивает возможность динамической и гибкой работы с моделью. Однако в ходе дальнейшего анализа сообщения и операции *должны* быть синхронизированы.

Когда линия жизни посылает сообщение, в ней находится *фокус управления* (*focus of control*), или *активация* (*activation*). По мере развития взаимодействия во времени активация перемещается между линиями жизни. Это движение называют *потоком управления* (*flow of control*).

Сообщения отображаются в виде стрелок между линиями жизни. Если линия жизни имеет пунктирный «хвост» (как на диаграммах последовательностей), сообщения обычно размещают между пунктирными линиями. В противном случае сообщения выстраиваются между пиктограммами линий жизни. В данной книге приводится множество примеров изображения сообщений. Существует семь типов сообщений (табл. 12.2).

Таблица 12.2

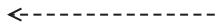
Синтаксис	Имя	Семантика
	Синхронное сообщение	Отправитель ожидает завершения выполнения сообщения получателем.
	Асинхронное сообщение	Отправитель посылает сообщение и продолжает исполнение – он *не* ожидает возврата от получателя.
	Возврат	Получатель сообщения возвращает фокус управления отправителю этого сообщения.
	Создание объекта	Отправитель создает экземпляр классификатора, определенного получателем.

Таблица 12.2 (продолжение)

Синтаксис	Имя	Семантика
	Уничтожение объекта	Отправитель уничтожает получателя. Если у линии жизни есть «хвост», он завершается символом X.
	Найденное сообщение	Отправитель сообщения находится вне области видимости взаимодействия. Используется, когда необходимо показать получение сообщения без указания его источника.
	Потерянное сообщение	Сообщение никогда не достигает точки своего назначения. Может использоваться для обозначения состояний ошибки, при которых пропадают сообщения.

12.7.1. Синхронные, асинхронные и сообщения возврата

При синхронном вызове сообщения отправитель ожидает завершения выполнения получателем запрашиваемой операции. При асинхронном отправитель ничего *не* ожидает, а переходит к следующему этапу.

Для аналитических моделей различие между синхронными и асинхронными сообщениями обычно является слишком высоким уровнем детализации. При анализе нас не интересует глубокая семантика процесса обмена сообщениями. Достаточно того факта, что сообщение отправлено. В этом случае все сообщения можно показывать как синхронные или асинхронные – это на самом деле не важно. Мы предпочитаем все сообщения отображать как синхронные, потому что это наиболее несвободный вариант. Синхронные сообщения показывают прямую последовательность вызовов операций, тогда как асинхронные сообщения указывают на возможный параллелизм.

При проектировании различие между синхронными и асинхронными сообщениями может иметь значение для создания параллельных потоков управления.

На уровне анализа реализаций прецедентов сообщение возврата можно показывать или не показывать – по желанию разработчика. Обычно они не имеют особого значения, и их наносят на диаграмму, только если это не загромождает ее.

12.7.2. Сообщения создания и уничтожения

В ОО анализе обычно нас не интересует конкретная семантика создания или уничтожения объекта, но понимать, что происходит, необходимо. Поэтому уделим внимание этому вопросу.

Сообщение создания объекта обычно изображается как сплошная линия с открытой стрелкой. Создание объекта можно показать с помощью сообщения со стереотипом «create» (создать). Или можно послать конкретное именованное сообщение создания объекта, которое также может быть обозначено стереотипом «create». В C++, C# или Java операции создания объектов являются специальными операциями, которые называют конструкторами. Имя конструктора совпадает с именем класса. Конструкторы не имеют возвращаемого значения, они могут иметь от нуля и более параметров. Например, для создания нового объекта Account можно было бы послать сообщение Account() и инициализировать его атрибут accountNumber некоторым значением. Однако конструкторы есть не во всех языках программирования. Например, в Smalltalk было бы послано сообщение «create» init: accountNumber.

Сообщение уничтожения объекта показывают сплошной линией с открытой стрелкой и стереотипом «destroy» (уничтожить). Уничтожение означает, что экземпляр классификатора, на который ссылается целевая линия жизни, больше не доступен для использования. Если у линии жизни есть «хвост», он должен завершаться большим крестом в точке уничтожения. Для уничтожения объектов нет возвращаемого значения.

В разных языках программирования семантика уничтожения различна. Например, в C++ уничтожение обычно явно обрабатывается программистом, и при уничтожении объекта гарантированно инициируется специальный метод (если он существует), называемый деструктором. Этот метод часто используется для проведения операций очистки, таких как высвобождение ресурсов, например файлов или соединений с базой данных. Вызов деструктора высвобождает память, выделенную под объект.

В таких языках программирования, как Java и C#, уничтожение объектов обрабатывается виртуальной машиной с помощью механизма под названием «сборка мусора». Например, если на объект Java-программы больше не ссылается ни один другой объект, он помечается как готовый к уничтожению. Уничтожение *произойдет* в некоторый момент времени в соответствии с алгоритмом сборки мусора, но вы не знаете, когда это случится! У объектов в Java и C# могут быть методы-«финализаторы», которые будут исполняться в момент реального уничтожения, осуществляемого сборщиком мусора. Однако использовать этот метод опасно, потому что мы точно не знаем, когда сборщик мусора вызовет его.

12.7.3. Найденные и потерянные сообщения

Обычно в анализе найденные и потерянные сообщения могут быть проигнорированы. Мы рассматриваем их здесь в основном для полноты обсуждения.

Найденные сообщения могут быть полезны, если необходимо показать получение сообщения классом, но неизвестно (в данный момент време-

ни), откуда поступило это сообщение. На практике такое встречается редко.

Потерянные сообщения позволяют показать, что сообщение потеряно – оно никогда не достигает точки своего назначения. Это может быть полезно при проектировании, чтобы показать, как могут теряться сообщения в условиях возникновения ошибки. Однако у нас никогда не возникало крайней необходимости использовать это понятие.

12.8. Диаграммы взаимодействий

Диаграммы взаимодействий UML могут использоваться для моделирования любого типа взаимодействия между экземплярами классификаторов. В частности, в реализации прецедентов диаграммы взаимодействий используются для моделирования взаимодействий между объектами, реализующими прецедент или его часть. Существует четыре разных типа диаграмм взаимодействий, каждый из которых делает акцент на различных аспектах взаимодействия.

> Четыре типа диаграмм взаимодействий предоставляют разные проекции взаимодействий объектов.

- Диаграммы последовательностей (sequence diagrams) акцентируют внимание на временной упорядоченности сообщений. Обычно пользователи лучше понимают диаграммы последовательностей, чем коммуникационные диаграммы, поскольку они намного легче читаются. Как правило, коммуникационные диаграммы очень быстро загромождаются. Диаграммы последовательностей обсуждаются в разделе 12.9.
- Коммуникационные диаграммы (communication diagrams) выделяют структурные отношения между объектами и очень полезны при анализе, особенно для создания эскиза совместной работы объектов. В UML 2 эти диаграммы предлагают только лишь подмножество функциональности диаграмм последовательностей. Коммуникационные диаграммы обсуждаются в разделе 12.11.
- Диаграммы обзора взаимодействий (interaction overview diagrams) показывают, как сложное взаимодействие реализуется рядом простых взаимодействий. Это особый случай диаграммы деятельности, в которой узлы ссылаются на другие взаимодействия. Они полезны для моделирования потока управления системы. Диаграммы обзора взаимодействий обсуждаются в разделе 15.12.
- Временные диаграммы (timing diagrams) обращают внимание на фактическое время взаимодействия. Их основное назначение – помочь оценить временные затраты. Временные диаграммы рассматриваются в разделе 20.7.

Диаграммы последовательностей и коммуникационные диаграммы являются самыми важными с точки зрения реализации прецедентов. Оставшаяся часть этой главы посвящена их детальному обсуждению.

12.9. Диаграммы последовательностей

Диаграммы последовательностей представляют взаимодействия между линиями жизни как упорядоченную последовательность событий. Это самая богатая и гибкая форма диаграммы взаимодействий.

> Диаграммы последовательностей представляют взаимодействия между линиями жизни как упорядоченную последовательность событий.

Иногда моделирование начинают с создания эскиза реализации прецедента с помощью коммуникационной диаграммы (раздел 12.11), потому что на диаграмме легко размещать и соединять линии жизни. Однако если необходимо сфокусировать внимание на установлении фактической *последовательности* событий, удобнее работать с диаграммой последовательностей.

12.9.1. Линии жизни и сообщения

Чтобы разобраться с линиями жизни и сообщениями, возьмем пример из простой системы регистрации курсов. Рассмотрим реализацию прецедента AddCourse (добавить курс) (рис. 12.6). Чтобы пример был простым, сохранен очень высокий уровень абстракции этого прецедента.

Прецедент: AddCourse
ID: 8
Краткое описание: Добавляет детали нового курса в систему.
Главные актеры: Registrar
Второстепенные актеры: Нет.
Предусловия: 1. Registrar вошел в систему.
Основной поток: 1. Registrar выбирает «add course». 2. Registrar вводит имя нового курса. 3. Система создает новый курс.
Постусловия: 1. Новый курс добавлен в систему.
Альтернативные потоки: CourseAlreadyExists

Рис. 12.6. *Спецификация прецедента AddCourse*

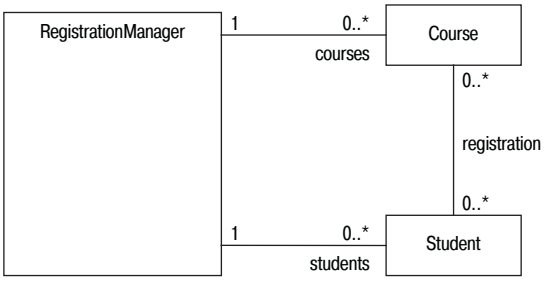


Рис. 12.7. *Диаграмма классов анализа прецедента AddCourse*

Исходный анализ созданного прецедента представлен на рис. 12.7 в виде высокоуровневой диаграммы классов анализа. Спецификация прецедента и диаграмма классов обеспечивают объем информации, достаточный для создания диаграммы последовательностей.

На рис. 12.8 показана диаграмма последовательностей, реализующая поведение, описанное прецедентом AddCourse. Согласно спецификации UML 2, имена диаграмм взаимодействий могут начинаться с приставки sd, для обозначения того, что данная диаграмма является диаграммой взаимодействий. Довольно странно: sd используется для *всех* типов диаграмм взаимодействий, а не только для диаграмм последовательностей (sd – sequence diagram)!

На данном этапе следует напомнить, что при создании диаграмм последовательностей как части реализации прецедента может обнаружиться необходимость доработки диаграммы классов анализа или даже прецедента. Это нормально, т. к. все это – часть процесса анализа.

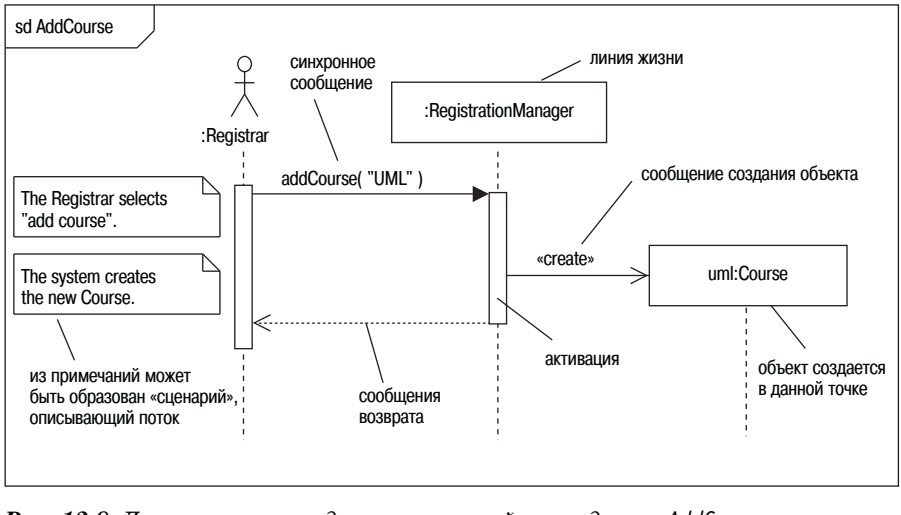


Рис. 12.8. *Диаграмма последовательностей прецедента AddCourse*

Прецедент, диаграмма классов анализа и диаграмма последовательностей – все они вместе эволюционируют со временем.

Рассмотрим диаграмму последовательностей, приведенную на рис. 12.8. Время на диаграммах последовательностей развивается сверху вниз, линии жизни выполняются слева направо. Линии жизни размещаются горизонтально, чтобы минимизировать число пересекающихся линий на диаграмме. Их местоположение относительно вертикальной оси отражает момент их создания. Пунктирная линия, находящаяся внизу, показывает существование линии жизни в течение времени.

> Диаграммы взаимодействий не являются дословными копиями прецедента; это иллюстрации того, как классы анализа реализуют поведение прецедента.

Обратите внимание, что рис. 12.8 показывает реализацию поведения прецедента, но не является точным представлением каждого его шага. Это важный момент. В шагах 1 и 2 участвует какой-то пользовательский интерфейс, которым не занимаются всерьез вплоть до этапа проектирования, поэтому на диаграмме последовательностей он опущен. Слой пользовательского интерфейса, который помогает прояснить некоторые вещи, может быть добавлен в диаграмму последовательностей во время проектирования (раздел 20.4). В анализе нас интересует только основное поведение классов анализа.

Давайте рассмотрим другой пример – прецедент DeleteCourse (удалить курс) из системы регистрации курсов (рис. 12.9).

В данном случае происходит уничтожение объекта. Уничтожение объекта обозначается большим крестом, завершающим линию жизни, как

Прецедент: DeleteCourse
ID: 8
Краткое описание: Удаляет курс из системы.
Главные актеры: Registrar
Второстепенные актеры: Нет.
Предусловия: 1. Registrar вошел в систему.
Основной поток: 1. Registrar выбирает «delete cource». 2. Registrar вводит имя курса. 3. Система удаляет курс.
Постусловия: 1. Курс удален из системы.
Альтернативные потоки: CourseDoesNotExist

Рис. 12.9. *Спецификация прецедента DeleteCourse*

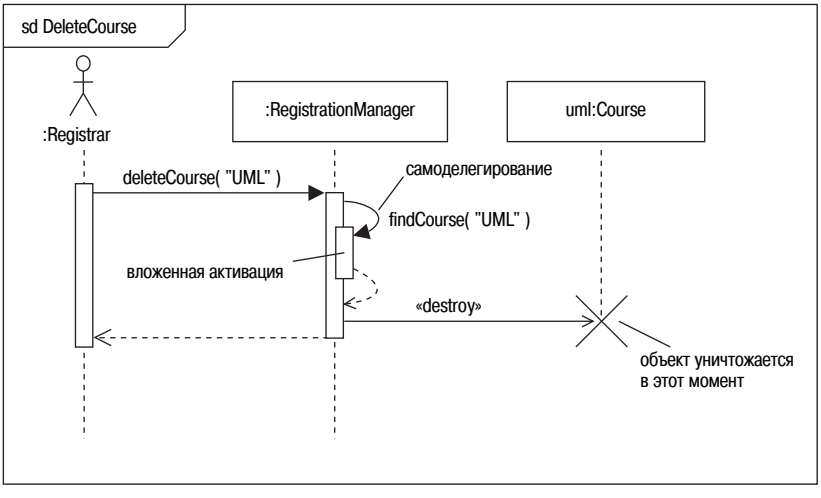


Рис. 12.10. *Самоделегирование*

показано на рис. 12.10. Если момент уничтожения объекта неизвестен или неважен, линия жизни завершается без креста.

На рис. 12.10 также представлено самоделегирование: линия жизни объекта посылает сообщение самой себе. Это создает вложенную активацию (см. следующий раздел). Самоделегирование распространено в ОО системах. Объекты предлагают ряд открытых сервисов (открытых операций), которые могут быть вызваны клиентскими объектами. Но, как правило, они также имеют и закрытые «вспомогательные» операции, разработанные специально для вызова самим объектом. В данном примере линия жизни :RegistrationManager посылает сама себе сообщение findCourse("UML"), чтобы найти UML-объект курса, если таковой существует. Закрытые операции объекта могут быть вызваны только самим объектом средствами самоделегирования.

12.9.2. Активации

Вытянутые прямоугольники, расположенные на пунктирной линии под линией жизни, показывают, когда на данной линии жизни находится фокус управления. Эти прямоугольники называются *активациями,* или *фокусом управления*.

> Активации показывают, когда фокус управления располагается в данной линии жизни.

Отметим, что в книге «The Unified Modeling Language Reference Manual, Second Edition» [Rumbaugh 1] активация определяется как «термин UML 1, замененный на термин “описание выполнения”». Однако нигде в «Unified Modeling Language: Superstructure, version 2.0» [UML2S]

термина «описание выполнения» («execution specification») нам найти не удалось, тогда как там есть и «активация», и «фокус управления». Отсюда можно сделать единственный вывод о том, что «активация» и «фокус управления» являются стандартными терминами. Мы вспомнили об этом, потому что в литературе может встретиться термин «описание выполнения» как синоним активации, поскольку он включен в книгу [Rumbaugh 1].

На рис. 12.8 фокус управления сначала находится на актере :Registrar. Он посылает сообщение addCourse("UML") актеру :RegistrationManager, который инициирует операцию addCourse(...) с параметром "UML". Во время выполнения этой операции фокус управления находится у :RegistrationManager. Однако обратите внимание, что этот фокус управления *вложен* в фокус управления актера :Registrar. Это вполне нормально – один объект, имеющий фокус управления, инициирует операцию другого объекта, создавая вложенный фокус управления. Затем этот объект может инициировать операцию еще одного объекта, еще более углубляя вложенность фокуса управления, и т. д.

В рамках выполнения операции addCourse(...) актер :RegistrationManager создает новый объект uml:Course.

В последние годы активации вышли из моды. Многие разработчики моделей не утруждают себя их отображением. Отчасти это объясняется плохой поддержкой активаций некоторыми инструментами UML, отчасти тем, что обычно активации не так важны, особенно в аналитических моделях. В исполняющейся ОО системе активации и так просматриваются через обычную семантику вызова операции. На практике читать сложные диаграммы последовательностей *без* активаций может быть немного проще. Мы используем активации, только если они не уменьшают читаемость диаграммы.

12.9.3. Документирование диаграмм последовательностей

Одно из замечательных свойств диаграмм последовательностей – возможность добавлять в них «сценарии» путем размещения примечаний в нижней левой части диаграмм (рис. 12.8). Это делает диаграмму намного более понятной заинтересованным лицам, которые не являются специалистами в UML, например пользователям или экспертам. Сценарий может состоять из фактических шагов прецедента или краткого обзора того, что происходит на диаграмме, представленного в текстовой форме. В любом случае сценарий может быть полезным дополнением, особенно если диаграмма сложна.

12.9.4. Инварианты состояния и ограничения

Сообщение, получаемое экземпляром, может обусловить изменение его состояния.

Состояние определяется как «условие или ситуация в ходе жизни объекта, в течение которой он удовлетворяет некоторому условию, осуществляет некоторую деятельность или ожидает некоторое событие» [Rumbaugh 1]. У каждого классификатора может быть автомат, описывающий жизненный цикл его экземпляров с точки зрения состояний, в которых они могут находиться, и событий, которые обуславливают переходы между этими состояниями.

Не все сообщения приводят к изменению состояния. Например, сообщение, которое возвращает значение некоторого атрибута и не имеет никаких побочных действий, никогда не генерирует изменения состояния. Состояние экземпляров на линиях жизни объектов можно показать с помощью *инвариантов состояния* (*state invariants*).

Добавление инвариантов состояния на диаграмму последовательностей может быть весьма полезным методом анализа, поскольку позволяет фиксировать ключевые состояния жизненного цикла линии жизни. Они отражают важные состояния системы и могут быть основой для автоматов, обсуждаемых в главе 21.

Рассмотрим конкретный пример. На рис. 12.11 показан прецедент из простой системы обработки заказов, имеющей следующие ограничения:

- заказ должен полностью оплачиваться *единовременным* платежом;
- позиции, указанные в заказе, могут поставляться только *после* оплаты;

Прецедент: ProcessAnOrder
ID: 5
Краткое описание: Покупатель делает заказ, который затем оплачивается и доставляется.
Главные актеры: Customer
Второстепенные актеры: Нет.
Предусловия: Нет.
Основной поток: 1. Прецедент начинается, когда актер Customer создает новый заказ. 2. Customer полностью оплачивает заказ. 3. Товары доставляются Customer в течение 28 дней после даты окончательного платежа.
Постусловия: 1. Заказ оплачен. 2. Товары доставлены в течение 28 дней после окончательного платежа.
Альтернативные потоки: ExcessPayment OrderCancelled GoodsNotDelivered GoodsDeliveredLate PartialPayment

Рис. 12.11. *Спецификация прецедента ProcessAnOrder*

- позиции доставляются покупателю в течение 28 дней после оплаты.

В реальности процесс обработки заказов обычно намного более сложный и гибкий по сравнению с рассматриваемым примером. Тем не менее пример служит иллюстрацией инвариантов состояний. На рис. 12.12 показана диаграмма последовательностей для ProcessAnOrder (обработка заказа).

Мы видим, что сразу после создания экземпляр Order переходит в состояние unpaid (не оплачено) (изображается в виде прямоугольника со скругленными углами). Это сообщает о том, что все вновь создаваемые экземпляры Order находятся в состоянии unpaid. При получении сообщения acceptPayment() (принять платеж), которое должно обозначать полную оплату, экземпляр переходит в состояние paid (оплачено). После этого экземпляр DeliveryManager (менеджер по доставке) посылает сообщение deliver() (доставить). Он передает это сообщение экземпляру Order, что приводит к его переходу в состояние delivered (доставлено).

Если класс Order также имеет автомат, его состояния должны соответствовать всем инвариантам состояний, присутствующим на его диаграммах последовательностей.

Рисунок 12.12 иллюстрирует применение ограничений. Они записываются в фигурных скобках и размещаются на (или рядом) линиях жизни. Ограничение, обозначенное на линии жизни, указывает на что-то, что должно быть истинным для экземпляров, начиная с этого момента и далее. Ограничения часто формулируются на естественном языке, хотя в UML есть формальный язык ограничений (OCL), который обсуждается в главе 25.

На рисунке показано ограничение продолжительности. Линия жизни объекта :Customer имеет две метки, A и B, и ограничение {B – A <= 28 days}.

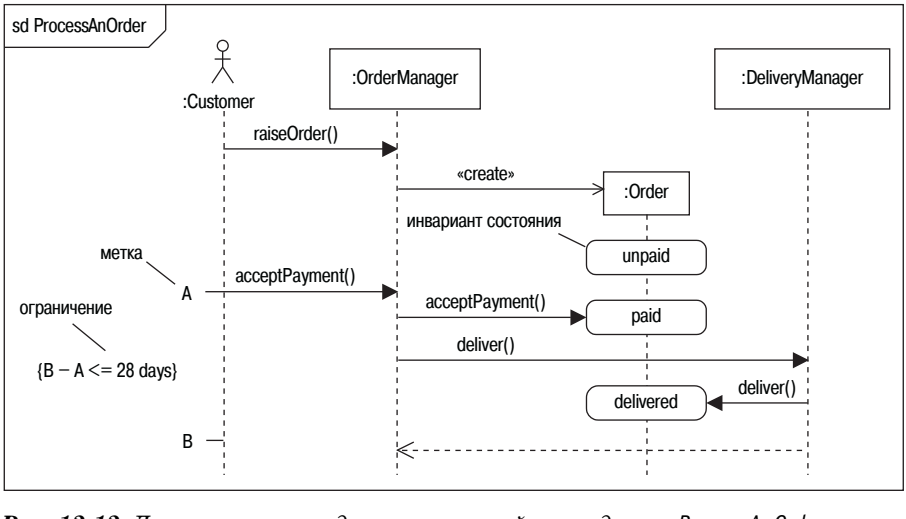


Рис. 12.12. *Диаграмма последовательностей прецедента ProcessAnOrder*

Это говорит о том, что промежуток времени между точками A и B не должен превышать 28 дней. Точка A обозначает момент оплаты, а точка B – момент доставки товаров покупателю. На естественном языке это ограничение означает, что «заказ должен быть доставлен в течение 28 дней после получения платежа».

На линию жизни может быть помещен любой тип ограничения. Широко распространены ограничения значений атрибутов экземпляров.

И наконец, обратите внимание, что на рис. 12.12 не использовались ни активации, ни возвраты сообщений. Это объясняется тем, что основное внимание данной диаграммы направлено на сроки и инварианты состояний, и упомянутые характеристики не добавили бы сюда никакой полезной информации.

12.10. Комбинированные фрагменты и операторы

Диаграммы последовательностей можно разделить на области, называемые *комбинированными фрагментами* (*combined fragments*). Рис. 12.13 иллюстрирует довольно богатый синтаксис комбинированного фрагмента.

> Комбинированные фрагменты разделяют диаграмму последовательностей на области с различным поведением.

У каждого комбинированного фрагмента есть один *оператор* (*operator*), один или более *операндов* (*operands*) и нуль или более *сторожевых условий* (*guard conditions*).

> Оператор определяет, *как* исполняются его операнды.

Оператор определяет, *как* исполняются его операнды.

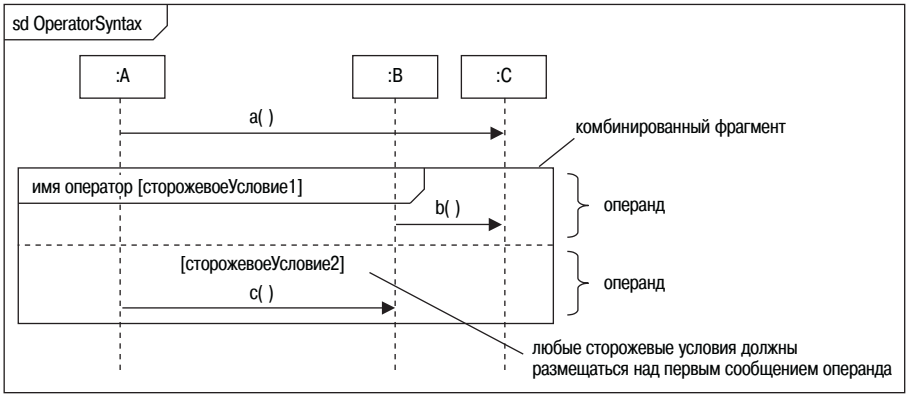


Рис. 12.13. *Синтаксис комбинированного фрагмента*

Сторожевые условия определяют, *будут ли* исполняться эти операнды. Сторожевое условие – это логическое выражение, и операнд исполняется тогда и только тогда, когда это выражение истинно. Одно сторожевое условие может применяться ко всем операндам или каждый операнд может иметь собственное уникальное сторожевое условие.

> Сторожевые условия определяют, *будут ли* выполняться данные операнды.

Полный список операторов приведен в табл. 12.3.

Таблица 12.3

Оператор	Полное имя	Семантика	Раздел
opt	option	Единственный операнд выполняется, если условие истинно (как if ... then).	12.10.1
alt	alternatives	Выполняется тот операнд, условие которого истинно. Вместо логического выражения может использоваться ключевое слово else (как select ... case).	12.10.1
loop	loop	Имеет специальный синтаксис: loop min, max [condition] повторять минимальное количество раз, затем, пока условие истинно, повторять еще (max – min) число раз.	12.10.2
break	break	Если сторожевое условие истинно, выполняется операнд, а *не* все остальное взаимодействие, в которое он входит.	12.10.2
ref	reference	Комбинированный фрагмент ссылается на другое взаимодействие.	13.2
par	parallel	Все операнды выполняются параллельно.	20.5.2
critical	critical	Операнд выполняется автоматически без прерывания.	20.5.2
seq	weak sequencing	Все операнды выполняются параллельно при условии выполнения следующего ограничения: последовательность поступления событий на *одну* линию жизни от *разных* операндов такая же, как и последовательность операндов. В результате наблюдается слабая форма упорядочения; отсюда название (weak sequencing – слабое упорядочение).	12.10
strict	strict sequencing	Операнды выполняются в строгой последовательности.	12.10

Таблица 12.3 (продолжение)

Оператор	Полное имя	Семантика	Раздел
neg	negative	Операнд демонстрирует неверные взаимодействия. Применяется, когда необходимо показать, что *не должно* произойти.	12.10
ignore	ignore	Перечисляет сообщения, которые намеренно исключены из взаимодействия. Разделенный запятыми список имен проигнорированных сообщений в фигурных скобках помещается после имени оператора, например {m1, m2, m3}. Например, взаимодействие может представлять тестовый пример, в котором принято решение игнорировать некоторые сообщения.	12.10
consider	consider	Перечисляет сообщения, намеренно включенные во взаимодействие. Разделенный запятыми список имен сообщений в фигурных скобках помещается после имени оператора. Например, взаимодействие может представлять тестовый пример, в который решено включить подмножество возможных сообщений.	12.10
assert	assertion	Операнд является единственно возможным допустимым поведением в данный момент взаимодействия, любое другое поведение было бы ошибочным. Используется как средство обозначения того, что некоторое поведение *должно* иметь место в определенной точке взаимодействия.	12.10

Самые важные из них: opt, alt, loop, break, ref, par и critical. Операторы opt, alt, loop и break подробно обсуждаются в следующих подразделах, а ref – в следующей главе. Операторы par и critical имеют отношение к параллелизму, что является вопросом проектирования. Они обсуждаются в разделе 20.5 при рассмотрении параллелизма. Остальные операторы используются редко. В таблице предоставлен достаточный объем информации для их применения в случае необходимости.

12.10.1. Ветвление с помощью операторов opt и alt

Рисунок 12.14 иллюстрирует синтаксис операторов opt и alt.

opt создает единственную ветвь.

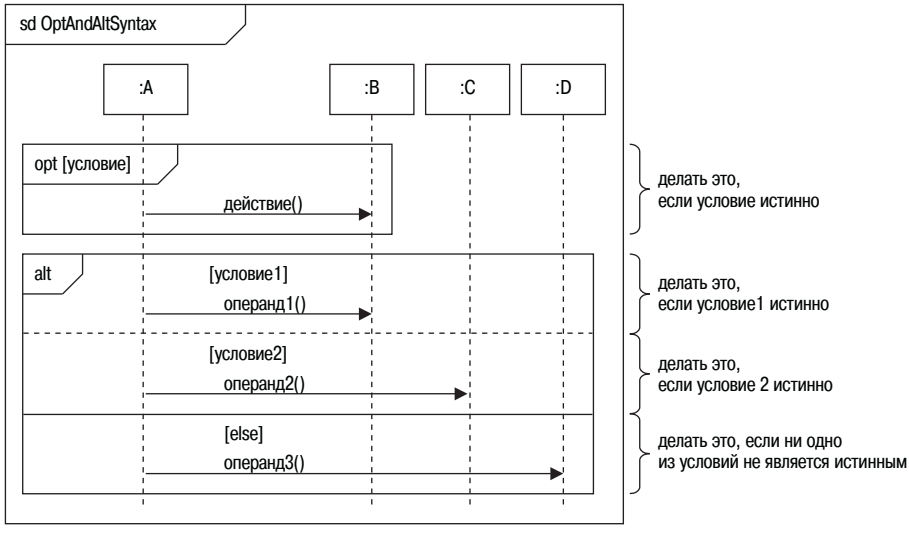


Рис. 12.14. *Синтаксис операторов opt и alt*

Оператор opt показывает, что его единственный операнд выполняется, если сторожевое условие истинно. В противном случае выполнение продолжается после комбинированного фрагмента.

opt эквивалентен программной конструкции:

```
if (условие1) then
  действие1
```

> **alt** создает несколько ветвей.

Оператор alt предоставляет выбор между рядом альтернатив. Каждый из его операндов имеет собственное сторожевое условие и будет выполняться, если только условие истинно. Необязательный операнд, заданный сторожевым условием [else], выполняется, если *ни одно* из других сторожевых условий не является истинным.

Это означает, что выполняться может *только один* из операндов alt, т. е. сторожевые условия всех операндов должны быть взаимоисключающими. Истинность более одного из сторожевых условий в любой момент времени является условием возникновения ошибки. Спецификация UML не описывает поведение alt в этом случае.

alt эквивалентен программной конструкции:

```
if (условие1) then
  операнд 1
else if (условие2) then
  операнд 2
```

Прецедент: ManageBasket
ID: 2
Краткое описание: Покупатель меняет количество товарных позиций в корзине.
Главные актеры: Customer
Второстепенные актеры: Нет.
Предусловия: 1. Содержимое корзины для покупок является видимым.
Основной поток: 1. Прецедент начинается, когда Customer выбирает товарную позицию в корзине. 2. Если Customer выбирает «delete item» 2.1. Система удаляет позицию из корзины. 3. Если Покупатель вводит новое количество 3.1. Система обновляет количество товаров в корзине.
Постусловия: Нет.
Альтернативные потоки: Нет.

Рис. 12.15. *Спецификация прецедента ManageBasket*

```
...
else if (условиеN) then
  операнд N
else
  операнд M
```

Как видите, оператор opt семантически эквивалентен оператору alt с одним операндом.

В качестве примера использования opt и alt рассмотрим прецедент на рис. 12.15. Впервые прецедент ManageBasket был показан в разделе 4.5.6. Это часть простой системы электронной коммерции. Он описывает процесс обновления количества объектов определенной товарной позиции или полного удаления этой позиции из корзины для покупок (ShoppingBasket) покупателя (Customer).

На рис. 12.16 показана диаграмма классов анализа этого прецедента. Как видите, в ShoppingBasket находится один или более Item. Каждый из этих Item – это quantity (количество) конкретного Product. На диаграмме классов анализа показывают только классы, атрибуты и операции, иллюстрирующие моделируемый момент. На данной диаграмме мы пытаемся показать совместную работу ShoppingBasket, Item и Product.

И наконец, на рис. 12.17 показана диаграмма последовательностей, реализующая этот прецедент. Обратите внимание, что на данной диаграмме описана стратегия, согласно которой Item уничтожается, когда его quantity достигает нуля. Это вполне обоснованно, поскольку Item существует только для того, чтобы представлять quantity конкретного

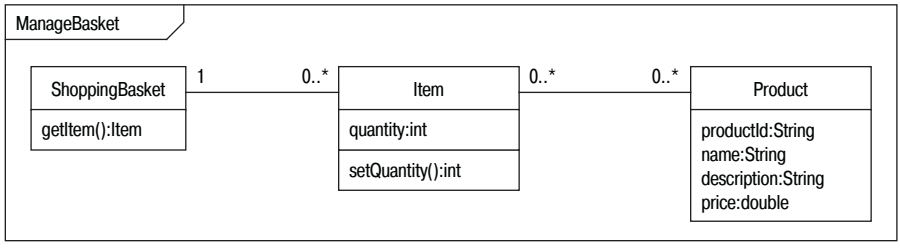


Рис. 12.16. *Диаграмма классов анализа прецедента ManageBasket*

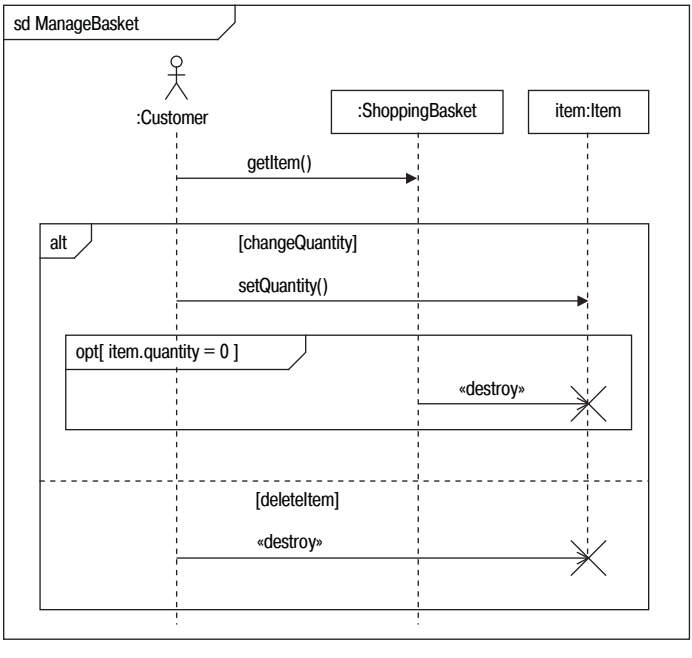


Рис. 12.17. *Диаграмма последовательностей прецедента ManageBasket*

Product в ShoppingBasket. Однако можно поспорить о необходимости такого уровня детализации в анализе. Диаграмма последовательностей, явно не отображающая удаление Item, тоже была бы вполне допустимой.

12.10.2. Организация итераций операторами loop и break

> **loop** позволяет моделировать итерацию.

Очень часто на диаграммах последовательностей необходимо показать циклы. Сделать это можно с помощью операторов loop и break, как показано на рис. 12.18.

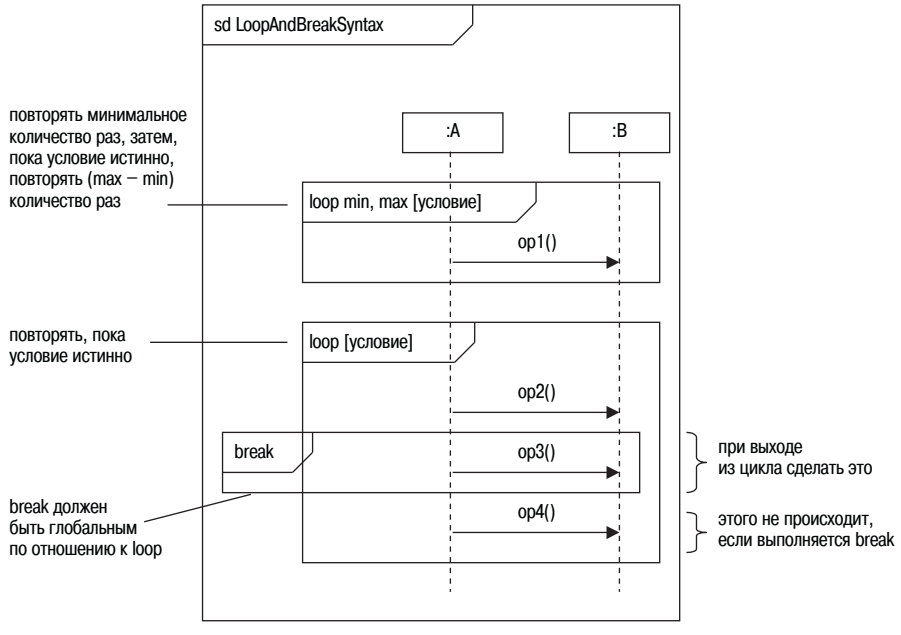


***Рис. 12.18.** Отображение циклов на диаграмме последовательностей с помощью операторов loop и break*

Оператор loop действует следующим образом:

```
loop min количество раз then
  while (условие истинно)
    loop (max - min) количество раз
```

В синтаксисе loop необходимо обратить внимание на следующие моменты:

- оператор loop *без* max, min или condition является бесконечным циклом;
- если задано только min, значит, max = min;
- condition обычно является логическим выражением, но может быть и произвольным текстом, смысл которого ясен.

На первый взгляд loop может показаться слишком сложным, но он может использоваться для поддержки огромного числа разнообразных циклов. Некоторые из самых распространенных циклов перечислены в табл. 12.4.

> Когда сторожевое условие **break** становится истинным, выполняется операнд **break**, цикл прерывается.

Оператор break может использоваться для обозначения условия прекращения loop и того, что происходит после этого. У оператора break

только одно сторожевое условие, и если оно истинно, исполняется тело break, а loop прерывается. Самое главное, что после завершения break loop *не* возобновляется.

Таблица 12.4

Тип цикла	Семантика	Выражение цикла
while(true) { тело }	Бесконечный цикл.	loop или loop *
for i = n to m { тело }	Повторяется (m – n) раз.	loop n, m
while(booleanExpression) { тело }	Повторяется, пока booleanExpression (логическое выражение) истинно.	loop [booleanExpression]

repeat
{ тело }
while(booleanExpression)
	Выполнить один раз, а затем повторять, пока booleanExpression истинно.	loop 1, * [booleanExpression]

forEach object in collection { тело }	Выполнить тело цикла по одному разу для каждого объекта в коллекции объектов.	loop [for each object in collectionOfObjects]
forEach object of class { тело }	Выполнить тело цикла по одному разу для каждого объекта определенного класса.	loop [for each object in ClassName]

Комбинированный фрагмент break логически находится вне цикла: он *не* является его частью. Поэтому фрагмент break всегда должен изображаться вне loop, но пересекаться с ним, как показано на рис. 12.18.

Самым распространенным применением циклов является перебор коллекции объектов. Например, цикл может применяться как возможная реализация операции findCourse(...) класса RegistrationManager (рис. 12.19).

Эта операция возвращает объект Course с соответствующим именем (name) или null, как показано на рис. 12.20. Имя конца ассоциации

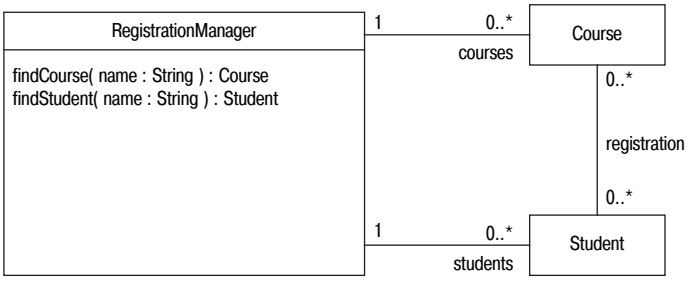


Рис. 12.19. *Применение цикла для реализации операции поиска FindCourse()*

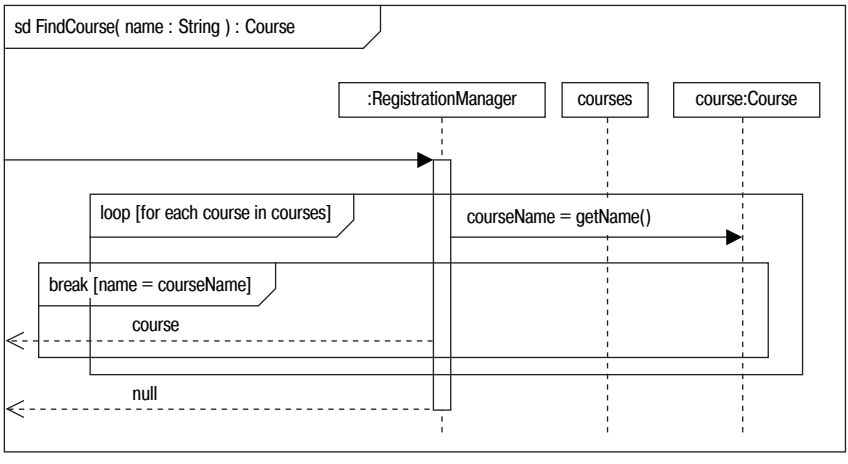


Рис. 12.20. *Применение цикла для перебора коллекции объектов*

с кратностью больше 1 можно использовать как коллекцию объектов. В данном случае для представления коллекции объектов Course используется имя courses (курсы).

12.11. Коммуникационные диаграммы

Коммуникационные диаграммы акцентируют внимание на структурных аспектах взаимодействия, на том, как соединяются линии жизни. В UML 2 их семантика довольно слаба по сравнению с диаграммами последовательностей.

> Коммуникационные диаграммы акцентируют внимание на структурных аспектах взаимодействия.

Мы уже приводили некоторые коммуникационные диаграммы. Согласно спецификации UML 2.0 [UML2S] диаграммы объектов, которые были представлены в главе 7, могут считаться особыми случаями диаграмм классов *или* особыми случаями коммуникационных диаграмм формы экземпляров, где каждая линия жизни представляет экземпляр класса (объект).

Синтаксис коммуникационных диаграмм подобен синтаксису диаграмм последовательностей, за исключением того, что здесь у линий жизни нет «хвостов». Вместо этого они соединены связями, образующими коммуникационные каналы для передачи сообщений. Порядок сообщений определяется путем иерархической нумерации.

На рис. 12.21 показана простая коммуникационная диаграмма для прецедента AddCourses, в котором :Registrar добавляет два новых объекта

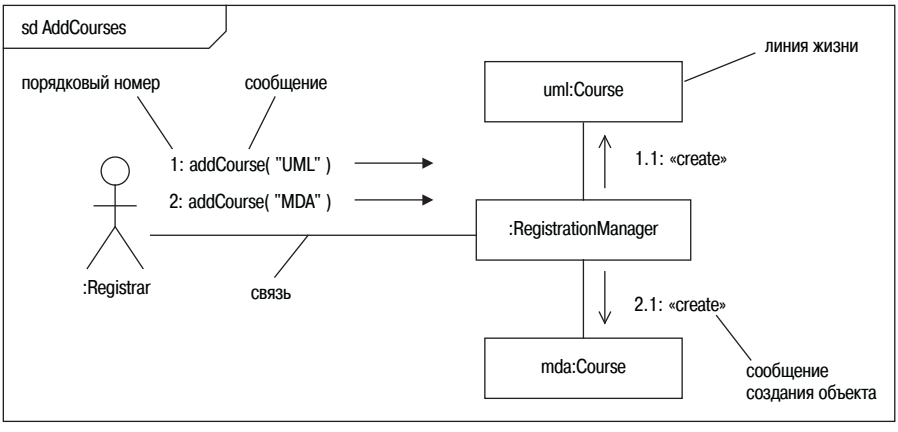


Рис. 12.21. *Коммуникационная диаграмма прецедента AddCourses*

Course. Следует обратить внимание на нумерацию сообщений, обозначающую их последовательность и вложенность в другие сообщения.

Ниже пошаговая интерпретация рис. 12.21.

1. addCourse("UML") – сообщение addCourse(…) отправляется линии жизни объекта :RegistrationManager с параметром "UML". Экземпляр :RegistrationManager инициирует операцию addCourse(…) и передает в нее параметр "UML", фокус управления переходит в эту операцию.

   1.1. «create» – порядковый номер 1.1 говорит о том, что фокус управления по-прежнему находится в операции addCourse(…). :RegistrationManager посылает анонимное сообщение, помеченное стереотипом «create». Такие сообщения «create» создают новые объекты, и это конкретное сообщение создает новый объект uml:Course. Позже при анализе или проектировании этому анонимному сообщению будет дано имя и, возможно, параметры, но пока достаточно показать, что создается новый объект uml:Course. После создания объекта из операции addCourse(…) больше не посылается никаких сообщений, и этот поток возвращается.

2. addCourse("MDA") – в :RegistrationManager посылается сообщение addCourse(…) с параметром "MDA". Фокус управления переходит к операции addCourse(…).

   2.1. «create» – порядковый номер 2.1 говорит о том, что фокус управления по-прежнему находится в операции addCourse(…). :RegistrationManager посылает анонимное сообщение, помеченное стереотипом «create»; это создает новый объект, mda:Course. После создания объекта фокус управления addCourse(…) возвращается и итерация завершается.

Поначалу могут возникнуть некоторые сложности с прочтением коммуникационных диаграмм, потому что в них заключено довольно мно-

го информации. Главное необходимо понимать, что в результате отправления сообщения вызывается некоторая операция экземпляра и что сложная нумерация сообщений указывает на вложенность вызовов операций (т. е. вложенный фокус управления).

12.11.1. Итерация

Показать итерацию на коммуникационных диаграммах можно с помощью выражения, описывающего итерацию. Оно включает спецификатор итерации (*) и (необязательный) блок итерации, как показано на рис. 12.22.

> Выражение, описывающее итерацию, определяет, сколько раз должно быть отправлено сообщение.

UML 2 не определяет никакого конкретного синтаксиса для блоков итераций, поэтому может использоваться все, что имеет смысл. Обычно хорошим вариантом являются код или псевдокод. Вероятно, вы думаете, что коммуникационные диаграммы могли бы по умолчанию использовать синтаксис итерации диаграммы последовательностей, описанный в разделе 12.10.2. Однако в спецификации UML об этом ничего не сказано! Если действительно применить аналогичный синтаксис итерации, описывающее итерацию сообщение могло бы выглядеть следующим образом:

```
* [ loop min, max [ условие ] ]
```

В такой записи есть преимущество единообразия, но и некоторая синтаксическая избыточность, поскольку и loop, и символ * являются спецификаторами итерации. Тем не менее мы считаем, что это очень хороший подход.

В примере на рис. 12.22 для обозначения повторения блока итерации при увеличении i от 1 до n применяется псевдокод. Затем i использует-

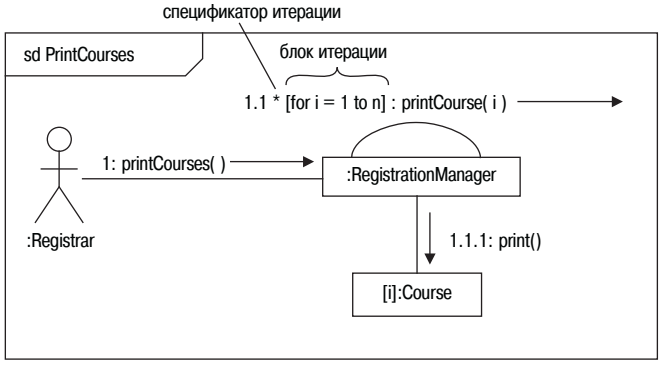


Рис. 12.22. *Синтаксис итерации на коммуникационных диаграммах*

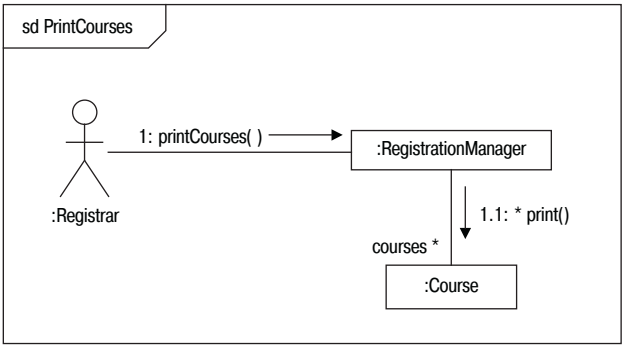


Рис. 12.23. *Альтернативный синтаксис итерации*

ся как селектор для выбора определенного экземпляра Course, которому отправляется сообщение print(). В результате этого происходит распечатка всех экземпляров Course. Однако такой подход предполагает, что экземпляры Course хранятся в индексированной коллекции. Если вы не хотите делать такое предположение, можно воспользоваться альтернативным подходом, показанным на рис. 12.23.

Рисунок 12.23 иллюстрирует два аспекта:

1. На связи между :RegistrationManager и :Course указано имя роли во множественном числе (courses) и кратность. Все это свидетельствует о том, что :RegistrationManager соединен с коллекцией объектов :Course (см. диаграмму классов на рис. 12.7).
2. К сообщению print() добавлен спецификатор итерации. Это указывает на то, что сообщение print() посылается *каждому объекту* коллекции.

Стандартный спецификатор итерации (*) означает, что сообщения будут выполняться последовательно. Если необходимо показать, что все сообщения выполняются параллельно, используется спецификатор параллельной итерации *//.

12.11.2. Ветвление

Ветвление можно смоделировать, добавив в сообщения сторожевые условия. Такие сообщения посылаются только тогда, когда сторожевое условие становится истинным.

> Ветвление – сообщение посылается, только если условие истинно.

На рис. 12.24 показан пример ветвления в нашей системе регистрации курсов. Эта коммуникационная диаграмма реализует прецедент RegisterStudentForCourse (зарегистрировать студента на курс). В этой системе регистрация является трехэтапным процессом:

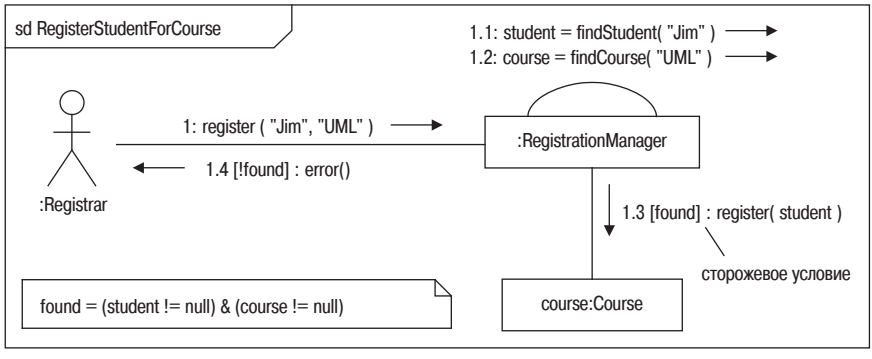


Рис. 12.24. *Коммуникационная диаграмма с ветвлением*

- найти запись студента – нельзя зарегистрировать студента на курс, если его нет в системе;
- найти необходимый курс;
- зарегистрировать студента на курс.

На рис. 12.24 широко представлены условия, чтобы продемонстрировать варианты их применения в коммуникационных диаграммах. Условия не имеют формального синтаксиса, но обычно являются выражениями, в которых используются временные переменные области действия текущего фокуса управления или атрибуты классов, участвующих во взаимодействии. На рис. 12.24 результаты операций findStudent(...) и findCourse(...) записываются в две временные переменные: student (студент) и course (курс). Затем значения этих переменных используются для вычисления значения временной булевой переменной found (найден). found применяется для образования ветви в шаге 1.3. Кроме того, found используется для принятия решения о формировании ошибки для :Registrar на шаге 1.4.

Рассмотрим пошаговую интерпретацию рис. 12.24.

1. registerStudent("Jim", "UML") – актер :Registrar посылает сообщение registerStudent("Jim", "UML") объекту :RegistrationManager.
   1.1. findStudent("Jim") – :RegistrationManager сам себе посылает сообщение findStudent("Jim"). Возвращаемое в результате этой операции значение сохраняется в переменной student. В случае неудачного поиска значение равно null.
   1.2. findCourse("UML") – :RegistrationManager сам себе посылает сообщение findCourse("UML"). Возвращаемое в результате этой операции значение сохраняется в переменной course. В случае неудачного поиска значение равно null.
   1.3. [found] register(student) – :RegistrationManager посылает сообщение register(student) объекту course. Это сообщение защищено условием и будет послано, если и student, и course nenull. Иначе

говоря, попытка зарегистрировать student на course делается только в случае успешного обнаружения обоих объектов, student и course.

1.4. [!found] : error() – если found имеет значение false, вызывается операция error() объекта :Registrar.

На коммуникационных диаграммах довольно сложно отчетливо показать ветвления – создается впечатление, что условия разбросаны по всей диаграмме; диаграмма очень быстро становится достаточно сложной. Основная рекомендация: показывайте на этих диаграммах только очень простое ветвление. Для представления сложных ветвлений больше подходят диаграммы последовательностей.

12.12. Что мы узнали

Реализация прецедентов – важнейшая часть процесса анализа. Она позволяет сравнить теорию с практикой, явно демонстрируя, как могут взаимодействовать объекты классов для обеспечения заданного поведения системы. Диаграммы взаимодействий показывают, как классы и объекты реализуют требования, определенные в прецеденте.

Мы изучили следующее:

- Реализации прецедентов создаются в деятельности UP Анализ прецедента; эта деятельность формирует часть динамического представления системы.
- Реализации прецедентов показывают, как взаимодействуют экземпляры классов анализа для реализации функциональных требований, определенных прецедентом.
  - Каждая реализация прецедента реализует только один прецедент.
  - Реализации прецедентов состоят из:
    - диаграмм классов анализа – они должны «рассказывать историю» об одном (или более) прецеденте;
    - диаграммы взаимодействий – должны демонстрировать, как взаимодействуют объекты для реализации поведения прецедента;
    - особых требований – во время реализации прецедента всегда выясняются новые требования и их необходимо фиксировать;
    - уточнений прецедента – вероятно, понадобится изменять прецедент во время его реализации.
- Взаимодействия – это единицы поведения контекстного классификатора.
  - Взаимодействия могут использовать любые возможности контекстного классификатора.

    - В реализации прецедента контекстный классификатор – это прецедент.
    - Общая форма диаграммы взаимодействий показывает взаимодействия между ролями, которые могут исполнять в этом взаимодействии экземпляры классификатора.
    - Форма экземпляров диаграммы взаимодействий показывает взаимодействия между конкретными экземплярами классификатора:
        - для линий жизни используется обычная нотация экземпляра.
- Линия жизни представляет участника взаимодействия – то, как экземпляр классификатора участвует во взаимодействии.
    - Каждая линия жизни имеет необязательное имя, тип и необязательный селектор.
    - Все линии жизни обозначаются той же пиктограммой, что и их тип.
    - Экземпляры обозначаются путем подчеркивания имени, типа и селектора.
- Сообщение представляет конкретный тип соединения между двумя взаимодействующими линиями жизни.
    - Синхронное сообщение (сплошная стрелка).
    - Асинхронное сообщение (открытая стрелка).
    - Возврат (открытая стрелка, пунктирная линия).
    - Сообщение создания (открытая стрелка, сплошная линия, стереотип «create»).
    - Сообщение уничтожения (открытая стрелка, сплошная линия, стереотип «destroy»).
    - Найденное сообщение (открытая стрелка, исходящая из заштрихованного кружка).
    - Потерянное сообщение (открытая стрелка, заканчивающаяся в заштрихованном кружке).
- Диаграммы взаимодействий.
    - Диаграммы последовательностей – акцентируют внимание на временной упорядоченности сообщений.
    - Коммуникационные диаграммы – акцентируют внимание на структурных отношениях между объектами.
    - Диаграммы обзора взаимодействий – акцентируют внимание на отношениях между взаимодействиями.
    - Временные диаграммы – акцентируют внимание на реальном времени взаимодействий.
- Диаграммы последовательностей.
    - Ход времени происходит сверху вниз.

- Линии жизни выполняются слева направо:
  - у линий жизни есть вертикальные пунктирные «хвосты», отражающие продолжительность существования линии жизни;
  - линии жизни могут иметь активации для обозначения того, что фокус управления находится на этой линии жизни;
  - линии жизни должны быть хорошо организованы, чтобы свести к минимуму количество пересекающихся линий.
- Поясняющие сценарии необходимо размещать на диаграмме последовательностей внизу слева.
- Инварианты состояний – символы состояний размещаются в соответствующих точках линии жизни.
- Ограничения заключаются в {} и размещаются на или рядом с линиями жизни.

- Комбинированные фрагменты – области разного поведения на диаграммах последовательностей.
  - Оператор определяет, *как* исполняются его операнды.
  - Сторожевое условие определяет, *будут ли* исполняться операнды.
  - Операнд заключает в себе поведение.
- Операторы.
  - opt – существует единственный операнд, который исполняется в случае истинности условия (как if … then).
  - alt – выполняется тот операнд, условие которого истинно.
  - loop – loop min, max [условие]:
    - loop или loop * – повторяется бесконечно;
    - loop n, m – повторяется (m - n) раз;
    - loop [booleanExpression] – повторяется, пока booleanExpression истинно;
    - loop 1, * [booleanExpression] – выполняется один раз, затем повторяется, пока booleanExpression истинно;
    - loop [for each object in collectionOfObjects] – тело цикла выполняется один раз для каждого объекта коллекции;
    - loop [for each object in className] – тело цикла выполняется один раз для каждого объекта класса.
  - break – если сторожевое условие истинно, выполняется операнд, а не остальная часть взаимодействия, в которую включен оператор.
  - ref – комбинированный фрагмент ссылается на другое взаимодействие.
  - par – все операнды исполняются параллельно.
  - critical – операнд исполняется атомарно без прерывания.

  - seq – операнды исполняются параллельно при условии выполнения следующего ограничения: последовательность поступления событий на одну линию жизни от разных операндов совпадает с последовательностью операндов.
  - strict – операнды исполняются в строгой последовательности.
  - neg – операнд показывает неверные взаимодействия.
  - ignore – перечисляет сообщения, намеренно исключенные из взаимодействия.
  - consider – перечисляет сообщения, намеренно включенные во взаимодействие.
  - assert – данный операнд является единственным действительным поведением в данный момент взаимодействия.
- Коммуникационные диаграммы – акцентируют внимание на структурных аспектах взаимодействия:
  - линии жизни соединены связями;
  - сообщения имеют порядковый номер – иерархическая нумерация соответствует вложенности фокуса управления.
- Итерация – в сообщении используются спецификатор итерации (*) и необязательный блок итерации.
  - Блок итерации определяет число повторений.
  - В блоке итерации может использоваться естественный язык, псевдокод, исходный код или нотация цикла (оператора loop) диаграммы последовательностей.
  - Итерацию по коллекции объектов можно обозначить, указав на целевом конце связи имя роли и кратность (>1) и предварив сообщение спецификатором итерации (*). Сообщение посылается всем объектам по очереди.
  - Спецификатор параллельной итерации *// применяется для обозначения параллельного выполнения сообщений.
- Ветвление – сообщение предваряется сторожевыми условиями. Сообщение исполняется, если сторожевое условие истинно.
  - Могут возникнуть сложности с четким представлением ветвления на коммуникационных диаграммах – для сложного ветвления используются диаграммы последовательностей.

13

Дополнительные аспекты реализации прецедентов

13.1. План главы

В этой главе представлены некоторые дополнительные возможности диаграмм взаимодействия, помогающие справляться с запутанностью диаграмм, обусловленной природой разрабатываемых систем. Усложнение диаграмм возникает и на стадии анализа, и при проектировании. Всегда нужно стремиться создавать максимально простые диаграммы взаимодействия, однако иногда это невозможно. В этих случаях надо попробовать методы, представленные в этой главе.

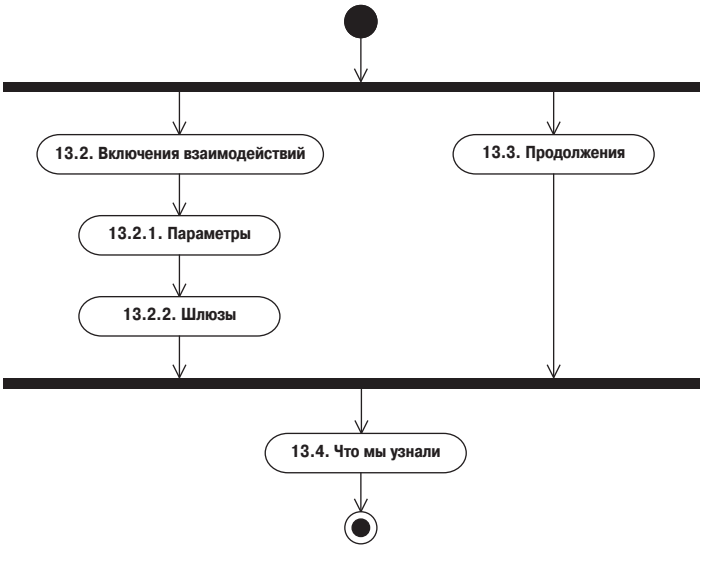


Рис. 13.1. *План главы*

13.2. Включения взаимодействий

Очень часто одна и та же последовательность сообщений многократно встречается в разных диаграммах последовательностей. Очевидно, что перерисовывать много раз один и тот же фрагмент взаимодействия – занятие утомительное и чреватое появлением ошибок. Поэтому в этих случаях применяются *включения взаимодействий*.

> Включения взаимодействий – это ссылки на другое взаимодействие.

Включения взаимодействий – это ссылки на взаимодействие. Когда такое включение помещается во взаимодействие, то в данной точке включается поток взаимодействия, на который ссылается данное включение взаимодействия.

Давайте в качестве примера рассмотрим фрагмент простой системы регистрации курсов, которая обсуждалась в главе 12. Диаграмма классов анализа для интересующей нас части системы представлена на рис. 12.7 (стр. 276).

Прежде чем актер Registrar (рис. 12.6) сможет работать с системой, он должен в нее войти. Это требование реализуется путем добавления на диаграмму (рис. 12.7) класса SecurityManager (менеджер безопасности), как показано на рис. 13.2.

Рассмотрим прецедент LogOnRegistrar (вход регистратора в систему), показанный на рис. 13.3. Он будет включен во все прецеденты, в которых Registrar сначала должен войти в систему.

Можно полагать, что фрагмент взаимодействия для входа Registrar в систему будет иметь место в начале многих диаграмм последовательностей. Имеет смысл выделить это общее поведение в отдельную диаграмму последовательностей и затем ссылаться на нее в случае необхо-

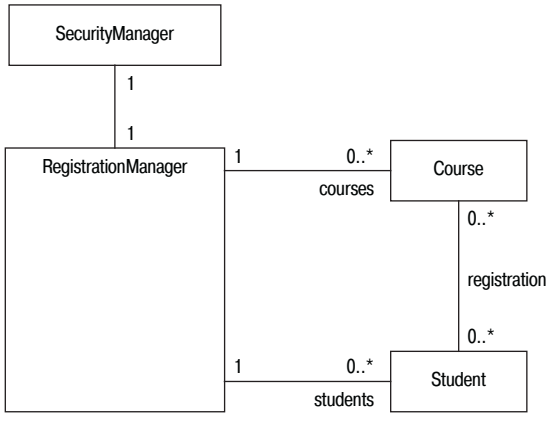


Рис. 13.2. *На диаграмму классов анализа добавлен класс SecurityManager*

Прецедент: LogOnRegistrar
ID: 4
Краткое описание: Registrar входит в систему.
Главные актеры: Registrar
Второстепенные актеры: Нет.
Предусловия: 1. Registrar еще не вошел в систему.
Основной поток: 1. Прецедент начинается, когда Registrar выбирает «log on». 2. Система спрашивает у Registrar имя пользователя и пароль. 3. Registrar вводит имя пользователя и пароль. 4. Система принимает имя пользователя и пароль как действительные.
Постусловия: 1. Registrar вошел в систему.
Альтернативные потоки: InvalidUserNameAndPassword RegistrarAlreadyLoggedOn

Рис. 13.3. *Спецификация прецедента LogOnRegistrar*

димости. На рис. 13.4 показана диаграмма последовательностей LogOnRegistrar, содержащая многократно используемый фрагмент взаимодействия.

На рис. 13.5 представлена другая диаграмма последовательностей, ChangeStudentAddress (изменить адрес студента), включающая взаимодействие LogOnRegistrar.

Вся последовательность событий ChangeStudentAddress представлена в табл. 13.1.

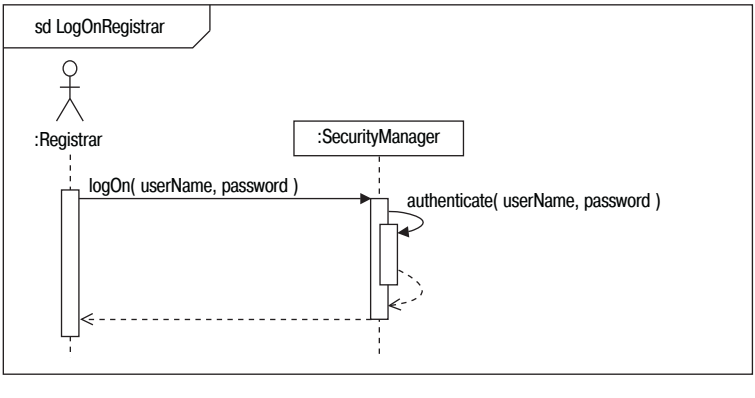


Рис. 13.4. *Диаграмма последовательностей, содержащая многократно используемый фрагмент взаимодействия*

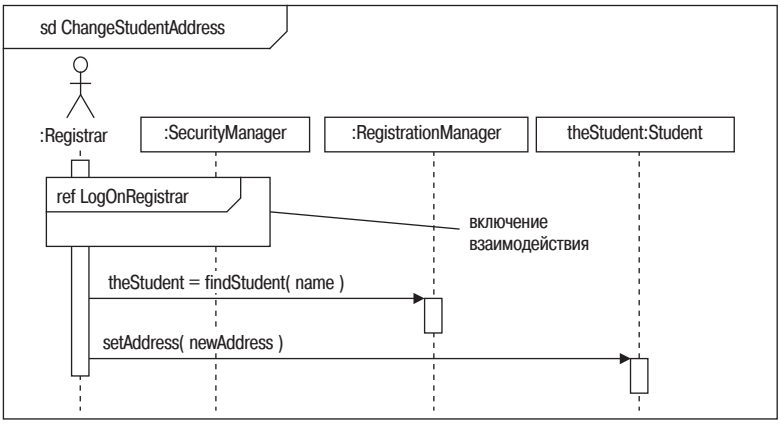


Рис. 13.5. *Диаграмма последовательностей с включением взаимодействия*

Таблица 13.1

Диаграмма	Исходная линия жизни	Сообщение	Целевая линия жизни
LogOnRegistrar	:Registrar	logOn(...)	:SecurityManager
LogOnRegistrar	:SecurityManager	authenticate(...)	:SecurityManager
ChangeStudentAddress	:Registrar	findStudent(...)	:RegistrationManager
ChangeStudentAddress	:Registrar	setAddress	theStudent:Student

Приведем несколько моментов, о которых необходимо помнить при использовании включений взаимодействия.

- Взаимодействие, на которое ссылается включение, вставляется во включающее взаимодействие в точке, где впервые появляется включение взаимодействия.
- По завершении включаемого взаимодействия необходимо быть очень внимательным к тому, где оказывается фокус управления! Следующее сообщение, отправляемое во включающем взаимодействии, должно быть согласовано с положением фокуса.
- Все линии жизни, участвующие во включении, должны присутствовать и во включающем взаимодействии.
- Чтобы обозначить область действия включения взаимодействия, оно отрисовывается поверх всех используемых им линий жизни.

13.2.1. Параметры

Взаимодействия могут иметь параметры. Это обеспечивает возможность поставлять взаимодействию разные значения в каждом включении взаимодействия. Параметры могут быть заданы с помощью обычного синтаксиса операций, рассмотренного в разделе 7.5.3.

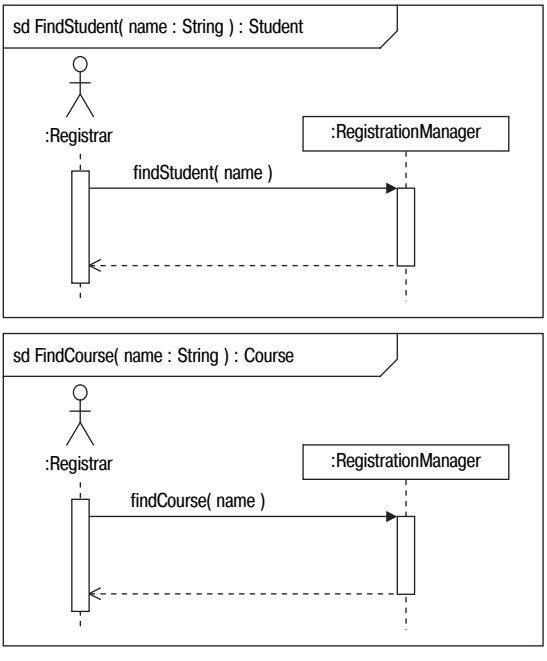


Рис. 13.6. *Параметризованные взаимодействия*

> Параметры дают возможность использовать конкретные значения во взаимодействии.

На рис. 13.6 показаны операции FindStudent(…) и FindCourse(…) – два параметризованных взаимодействия.

На рис. 13.7 показан пример использования параметризованных взаимодействий. Обратите внимание на то, как можно передавать во взаи-

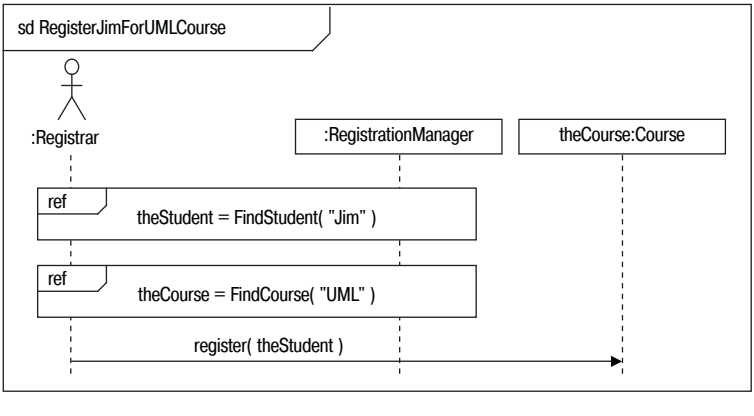


Рис. 13.7. *Передача конкретных значений в виде параметров*

модействия конкретные значения в виде параметров. Это обеспечивает огромные возможности и гибкость!

На рис. 13.7 можно увидеть, что два варианта употребления взаимодействий возвращают значения, присваиваемые временным переменным theStudent (студент) и theCourse (курс). Эти временные переменные существуют в области действия диаграммы последовательностей.

13.2.2. Шлюзы

Шлюзы (gates) – это входы и выходы взаимодействий.

> Шлюзы – это входы и выходы взаимодействий.

Шлюзы используются, когда взаимодействие необходимо инициировать линией жизни, которая *не* является частью взаимодействия. Воспользуемся операциями FindStudent и FindCourse на рис. 13.6, чтобы проиллюстрировать применение шлюзов (рис. 13.8).

Как видно из рисунка, шлюз – это точка на рамке диаграммы последовательностей, соединяющая сообщение, находящееся *вне* рамки, с сообщением *внутри* рамки. Сигнатуры обоих сообщений *должны* быть одинаковыми.

Рисунок 13.7 можно изменить, применив эти новые диаграммы последовательностей со шлюзами, как показано на рис. 13.9.

Теперь у FindStudent и FindCourse есть явные входы и выходы. Эти взаимодействия даже стали еще более гибкими, чем раньше. Рассмотрим рис. 13.10, на котором показан другой возможный вариант применения взаимодействия FindCourse.

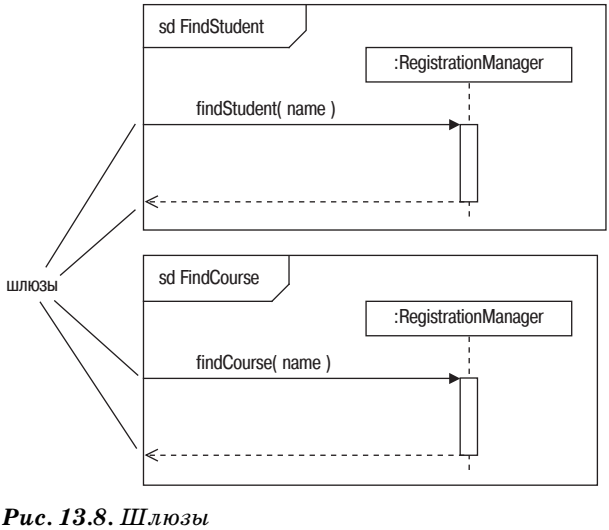


***Рис. 13.8.** Шлюзы*

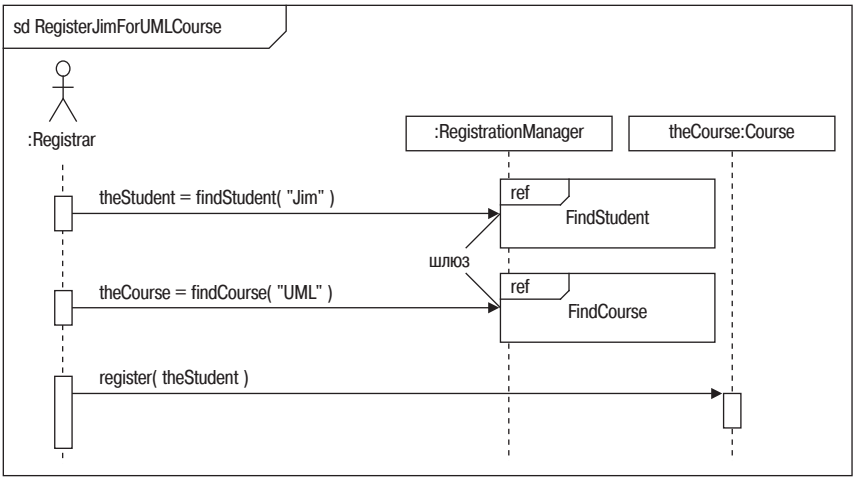


Рис. 13.9. *Диаграммы последовательностей с шлюзами*

И шлюзы, и параметры обеспечивают гибкость в многократном использовании взаимодействий. Возникает вопрос: когда должны применяться шлюзы, а когда – параметры?

- Параметры используются, когда известны исходные и целевые линии жизни всех сообщений взаимодействия.
- Шлюзы используются, когда некоторые сообщения приходят из-за границ рамки взаимодействия и заранее неизвестно, откуда они могут поступить.

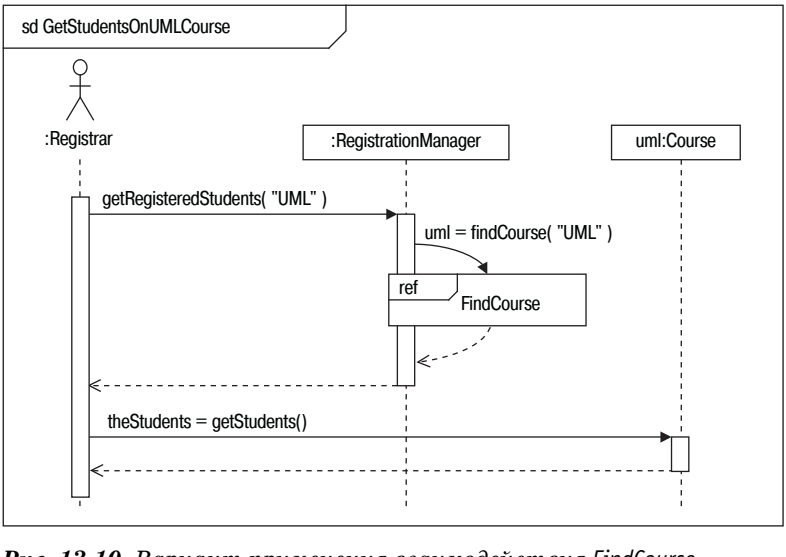


Рис. 13.10. *Вариант применения взаимодействия FindCourse*

13.3. Продолжения

Продолжения (continuations) позволяют фрагменту взаимодействия показать, что его поток завершается таким образом, что он может быть подхвачен и продолжен другим фрагментом взаимодействия. Продолжение изображается в виде метки внутри прямоугольника со скругленными углами.

> Продолжения обеспечивают возможность завершать фрагмент взаимодействия таким образом, что его может продолжить другой фрагмент.

Когда продолжение является *последним* элементом фрагмента взаимодействия, оно обозначает точку, в которой этот фрагмент завершается, но может быть продолжен другими фрагментами.

Если продолжение является *первым* элементом фрагмента взаимодействия, оно показывает, что этот фрагмент является продолжением предыдущего фрагмента.

Продолжения обеспечивают способ соединения разных взаимодействий. По сути, одно взаимодействие завершается, оставляя свои линии жизни в определенном состоянии, а другие взаимодействия могут присоединиться в этой точке и продолжить работу.

Визуальный синтаксис продолжений аналогичен инвариантам состояния, которые обсуждались в разделе 12.9.4. Однако продолжение – это лишь способ соединения разных взаимодействий в помеченных точках. Оно не обязательно отображается в конкретное состояние автомата контекстного классификатора.

На рис. 13.11 показана простая диаграмма последовательностей, на которой :RegistationUI (UI – пользовательский интерфейс) вызывает актера :Registrar сначала для получения имени курса, а затем для осуществления одной из трех операций: добавить, удалить или найти. В зависимости от того, какой вариант выбран, взаимодействие заканчивается одним из трех продолжений: addCourse, removeCourse или findCourse.

На рис. 13.12 можно увидеть взаимодействие HandleCourseOption (произвести действие над курсом), которое включает GetCourseOption (получить вариант действия над курсом) и затем выбирает одно из его продолжений. Как видите, продолжения позволяют:

- разъединить взаимодействия GetCourseOption и HandleCourseOption;
- потенциально повторно использовать GetCourseOption и HandleCourseOption с другими взаимодействиями, имеющими аналогичные продолжения.

При использовании продолжений необходимо помнить следующее:

- Продолжения начинают и заканчивают взаимодействия, следовательно, они должны быть или первым, или последним элементом взаимодействия.

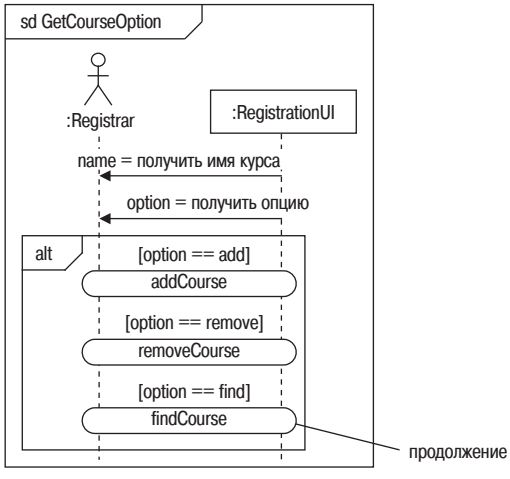


Рис. 13.11. *Диаграмма последовательностей с тремя продолжениями*

- В контексте данного классификатора продолжения под одним именем *должны* охватывать один набор линий жизни.
- Продолжения имеют смысл, только если во взаимодействии присутствует хотя бы слабое упорядочение – очевидно, что если нет упорядочения, неизвестно, где будет находиться продолжение.
- Продолжения должны охватывать *все* линии жизни включающего их фрагмента (т. е. они являются *глобальными* в рамках этого фрагмента).

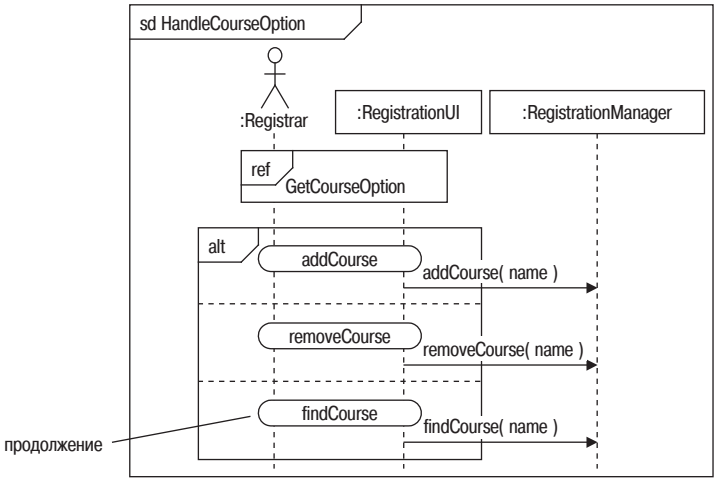


Рис. 13.12. *Взаимодействие HandleCourseOption включает GetCourseOption и затем выбирает одно из его продолжений*

Продолжения часто используются с оператором alt, как показано на рис. 13.12, для создания точек ветвления во взаимодействии.

13.4. Что мы узнали

В этой главе были рассмотрены дополнительные возможности диаграмм взаимодействия.

Мы изучили следующее:

- Включения взаимодействий – это ссылки на другое взаимодействие.
  - Поток используемого взаимодействия включается в поток использующего его взаимодействия.
  - Параметры – включения взаимодействия могут иметь параметры; используется обычная нотация параметров.
  - Шлюзы – входы и выходы взаимодействий:
    - точка на рамке диаграммы последовательностей, соединяющая сообщение, находящееся вне рамки, с сообщением, находящимся внутри рамки; сигнатуры обоих сообщений *должны* быть одинаковыми.
  - Параметры используются, когда известны источники и цели всех сообщений; в противном случае используются шлюзы.
- Продолжения – завершают фрагмент взаимодействия таким образом, чтобы его мог продолжить другой фрагмент:
  - если продолжение – первый элемент фрагмента, то фрагмент продолжает другой фрагмент;
  - если продолжение – последний элемент фрагмента, то фрагмент завершается, но может быть продолжен другим фрагментом.

14

Диаграммы деятельности

14.1. План главы

Диаграммы деятельности – это «ОО блок-схемы». Они позволяют моделировать процесс как деятельность, состоящую из коллекции соединенных ребрами узлов. UML 2 вводит новую семантику диаграмм деятельности, которая обеспечивает им намного большую мощь и гибкость, чем ранее. В этой главе рассматриваются основы диаграмм деятельности в объеме, достаточном для моделирования деятельности. Более глубокие вопросы рассматриваются в следующей главе.

14.2. Что такое диаграммы деятельности

Диаграммы деятельности часто называют «ОО блок-схемами». Они позволяют моделировать процесс как деятельность, которая состоит из коллекции соединенных ребрами узлов.

В UML 1 диаграммы деятельности фактически были лишь особым случаем диаграмм состояний (глава 21), где у каждого состояния было входное действие, которое определяло некоторый процесс или функцию, имеющие место при входе в состояние. В UML 2 диаграммы деятельности имеют совершенно новую семантику, базирующуюся на технологии сетей Петри (Petri Nets). В использовании этой технологии есть два преимущества:

1. Формализм сети Петри обеспечивает большую гибкость при моделировании различных типов потока.
2. В UML теперь есть четкое разделение между диаграммами деятельности и диаграммами состояний.

> Диаграммы деятельности – это ОО блок-схемы.

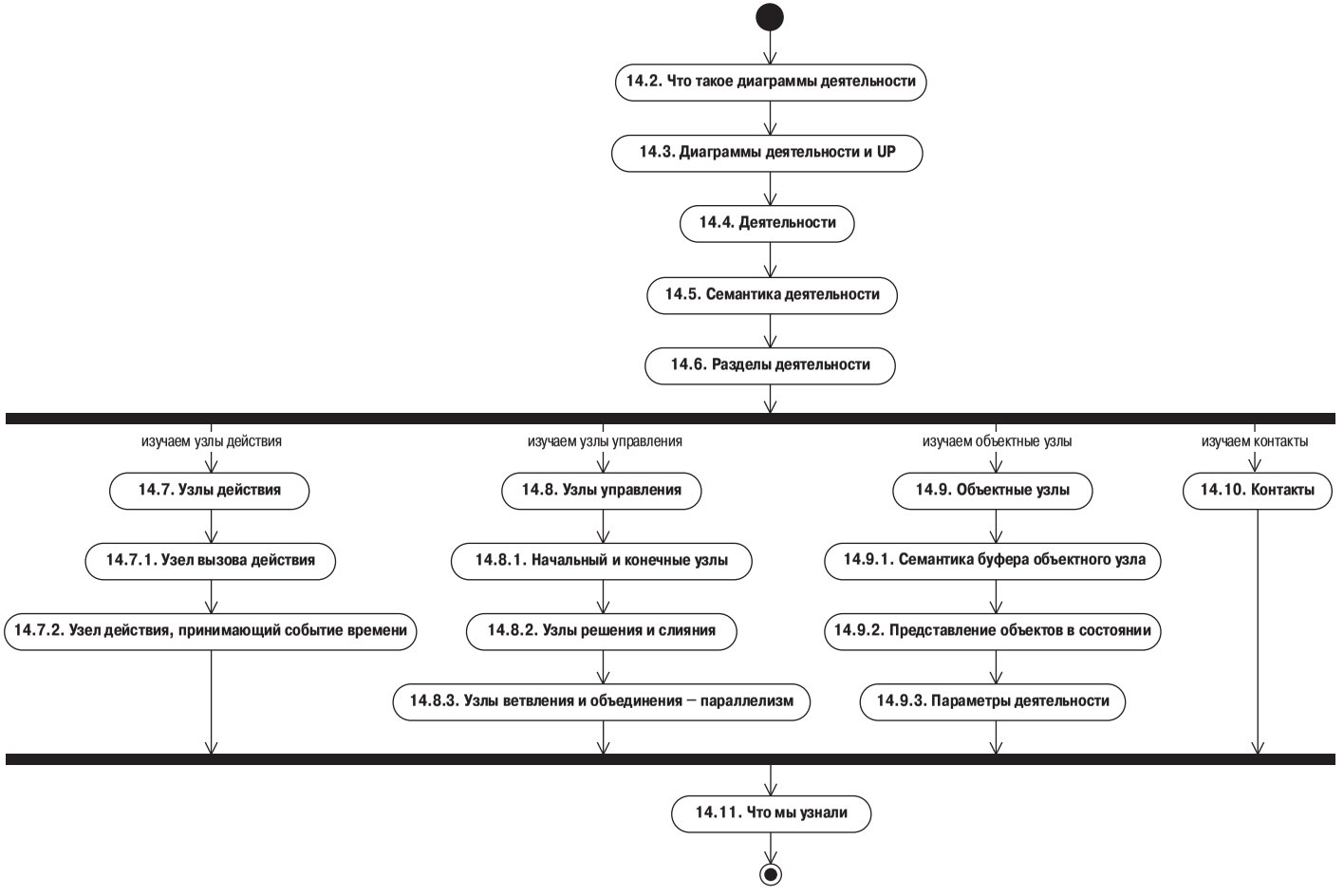


Рис. 14.1. *План главы*

Деятельность может быть добавлена к *любому* элементу модели с целью моделирования его поведения. Элемент обеспечивает контекст для деятельности, и деятельность может использовать возможности своего контекста. Деятельности обычно добавляются к:

- прецедентам;
- классам;
- интерфейсам;
- компонентам;
- кооперациям;
- операциям.

Диаграммы деятельности также могут использоваться для моделирования бизнес-процессов и рабочих потоков. Мы кратко покажем, как это делать, но более сложные аспекты выходят за рамки этой книги.

Хотя диаграммы деятельности обычно используются как блок-схемы операций, следует отметить, что исходный код операции в виде кода или псевдокода, возможно, является лучшим и более кратким их представлением! Так что о каждом случае необходимо судить отдельно.

Хорошая диаграмма деятельности сосредоточена на отражении лишь одного определенного аспекта динамического поведения системы. Таким образом, она должна находиться на соответствующем уровне абстракции, чтобы донести эту идею до целевой аудитории, и содержать минимум необходимой информации. Диаграммы деятельности можно дополнять состояниями и потоками объектов, но необходимо постоянно спрашивать себя, проясняют ли эти элементы диаграмму или делают ее еще более запутанной? Как обычно, лучше придерживаться максимальной простоты.

14.3. Диаграммы деятельности и UP

> Диаграммы деятельности могут использоваться во многих рабочих потоках UP.

Благодаря своей гибкости диаграммы деятельности не имеют одного определенного назначения в UP. Они обеспечивают универсальный механизм моделирования поведения и могут использоваться везде, где возникает необходимость в их применении. Мы обсуждаем их в рабочем потоке анализа, потому что чаще всего эти диаграммы используются при анализе.

Уникальная способность диаграмм деятельности в том, что они позволяют моделировать процесс *без* необходимости определения статической структуры классов и объектов, реализующих процесс. Несомненно, это очень полезно на ранних этапах анализа при попытках выяснить, что представляет собой конкретный процесс.

Из собственного опыта можем сказать, что диаграммы деятельности чаще всего используются в следующих случаях.

- В процессе анализа:
  - для графического моделирования потока прецедента. Такое представление является более понятным для заинтересованных сторон;
  - для моделирования потока между прецедентами. При этом используется особая форма диаграммы деятельности – диаграмма обзора взаимодействий (раздел 15.12).
- При проектировании:
  - для моделирования деталей операции;
  - для моделирования деталей алгоритма.
- При моделировании деловой активности:
  - для моделирования бизнес-процесса.

Обычно заказчикам проще понимать диаграммы деятельности, поскольку большинство из них имеет представление о блок-схемах в той или иной форме. Следовательно, диаграммы деятельности могут быть замечательным средством общения, если *они просты*.

Как будет видно в этой и следующей главах, UML 2 представляет много мощных нововведений в синтаксисе и семантике диаграмм деятельности. Важно не слишком сильно увлекаться всем этим. При создании *любой* UML-диаграммы всегда нужно помнить о целевой аудитории и использовать возможности UML соответственно. Нет смысла применять все самые последние нововведения, если никто не поймет диаграмму.

14.4. Деятельности

Деятельности – это системы *узлов* (*nodes*), соединенных *ребрами* (*edges*). Существует три категории узлов:

1. Узлы действия (action nodes) – представляют отдельные единицы работы, элементарные *в рамках деятельности*;
2. Узлы управления (control nodes) – управляют потоком деятельности;
3. Объектные узлы (object nodes) – представляют объекты, используемые в деятельности.

Ребра представляют потоки деятельности. Существует два типа ребер:

1. Ребра потоков управления (control flows) – представляют поток управления деятельности;

> Деятельности – это системы узлов, соединенных ребрами.

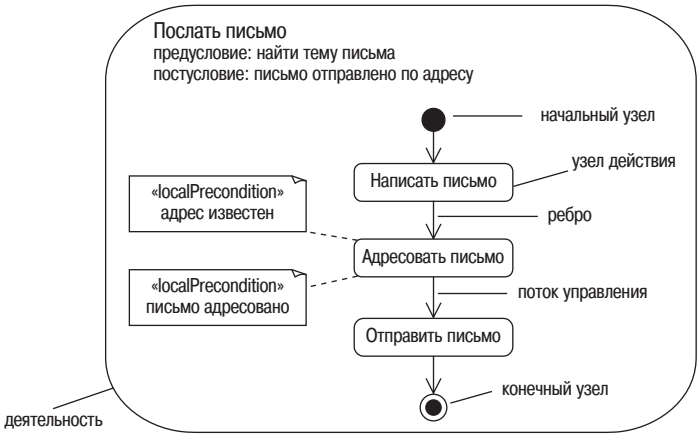


Рис. 14.2. *Диаграмма деятельности бизнес-процесса Послать письмо*

2. Ребра потоков объектов (object flows) – представляют поток объектов деятельности.

Все типы узлов и ребер будут подробно изучены в последующих разделах.

Давайте рассмотрим пример. На рис. 14.2 показана простая диаграмма деятельности бизнес-процесса Послать письмо. Обратите внимание, что деятельности могут иметь предусловия и постусловия, как и прецеденты. Предусловия – это условия, которые должны быть истинными, чтобы деятельность могла начаться, а постусловия – это условия, которые будут истинными по завершении деятельности. Действия внутри деятельности тоже могут иметь собственные локальные предусловия и постусловия, как показано на рис. 14.2.

Деятельности обычно начинаются с одного узла управления, начального. Он обозначает начало исполнения при вызове деятельности. Один или более конечных узлов показывают места завершения деятельности.

В примере, изображенном на рис. 14.2, деятельность начинается в начальном узле. Затем управление переходит вдоль ребра к узлу действия Написать письмо. Этот узел обозначает элементарную часть работы или поведения, поскольку содержит описываемую деятельность. Поток проходит через узлы Адресовать письмо, Отправить письмо и затем к конечному узлу, в котором деятельность завершается.

Обычно диаграммы деятельности используются для моделирования прецедента в виде последовательностей действий. На рис. 14.3 показан прецедент PaySalesTax (выплата налога с оборота) из главы 4. Этот прецедент может быть представлен в виде диаграммы деятельности, как показано на рис. 14.4.

Прецедент: PaySalesTax
ID: 1
Краткое описание: Выплата налога с оборота в налоговое управление по окончанию налогового периода.
Главные актеры: Time
Второстепенные актеры: TaxAuthority
Предусловия: 1. Конец налогового периода.
Основной поток: 1. Прецедент начинается в конце налогового периода. 2. Система определяет сумму налога с оборота, которую необходимо выплатить TaxAuthority. 3. Система посылает электронный платеж в TaxAuthority.
Постусловия: 1. TaxAuthority получает соответствующую сумму налога с оборота.
Альтернативные потоки: Нет.

Рис. 14.3. *Спецификация прецедента PaySalesTax*

Обратите внимание, что диаграмма деятельности – это более компактная и графическая форма прецедента. Диаграмма деятельности представляет прецедент как два действия: Вычисление налога с оборота и Отправка электронного платежа. Каждое из этих действий могло бы быть представлено в виде отдельной диаграммы деятельности. Вероятно, так и произойдет при проектировании, когда понадобится показать способ реализации этих действий. Актер и его взаимодействие с системой – это структурные элементы, поэтому их нет на этой диаграмме.

Прецеденты представляют поведение системы как взаимодействие актеров и системы, тогда как диаграммы деятельности отображают его в виде последовательности действий. Они являются дополнительными представлениями этого поведения.

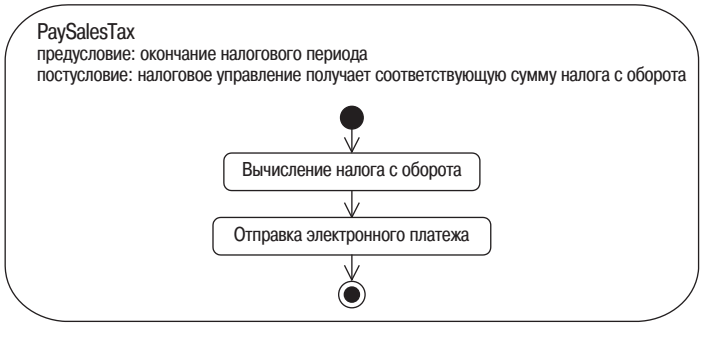


Рис. 14.4. *Диаграмма деятельности прецедента PaySalesTax*

14.5. Семантика деятельности

> Диаграммы деятельности основаны на технологии сетей Петри.

Семантика диаграмм деятельности, как следует из предыдущего изложения, интуитивно довольно проста. Данный раздел посвящен подробному описанию семантики деятельности.

Диаграммы деятельности UML 2 основаны на технологиях сетей Петри. Сети Петри выходят за рамки нашей книги. Более подробную информацию о них можно найти по адресу *www.daimi.au.dk/PetriNets/*.

Диаграммы деятельности моделируют поведение с помощью *«игры» маркеров* (*token game*). Эта игра описывает поток маркеров, движущийся по сети узлов и ребер согласно определенным правилам. Маркеры на диаграммах деятельности UML могут представлять:

- поток управления;
- объект;
- некоторые данные.

Состояние системы в любой момент времени определяется расположением ее маркеров.

В примере на рис. 14.2 маркер – это поток управления, поскольку в рассматриваемом случае между узлами не происходит передачи объектов или данных.

Маркеры перемещаются вдоль ребра (edge) от начального узла (source node) к целевому узлу (target node). Перемещение маркера происходит только при выполнении *всех* необходимых условий. Условия меняются в зависимости от типа узла. Для узлов, представленных на рис. 14.5 (узлы действия), этими условиями являются:

- постусловия начального узла;
- сторожевые условия ребра;
- предусловия целевого узла.

Условия существуют не только для узлов действия, но и для узлов управления и объектных узлов. Узлы управления имеют особую семантику, которая управляет тем, как передаются маркеры от входных ребер к выходным. Например, начальный узел начинает деятельность, конечный узел (final node) завершает деятельность, а узел объединения будет предлагать маркер на своем единственном исходящем ребре только в случае, если маркеры поступили на все его входные ребра. Объектные узлы представляют объекты, существующие в системе. Узлы управления и объектные узлы подробно обсуждаются в разделах 14.8 и 14.9 соответственно.

Рассмотрим «игру» маркеров для деятельности, показанной на рис. 14.5. С началом выполнения деятельности в начальном узле стартует поток

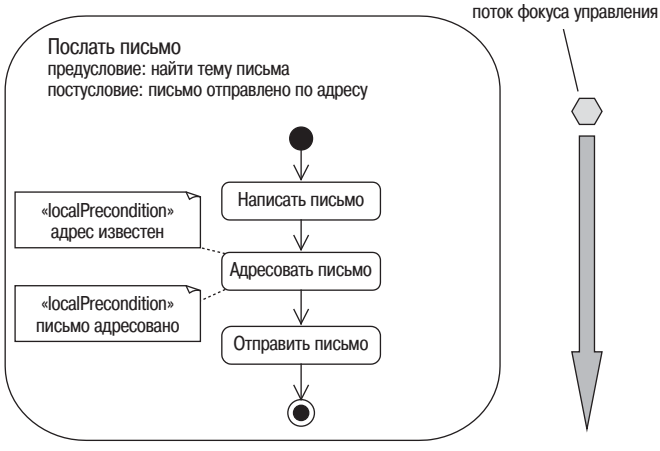


Рис. 14.5. *Диаграмма деятельности Послать письмо*

маркера управления. Ни на начальный узел, ни на его выходное ребро, ни на целевой узел не наложено никаких условий, поэтому маркер автоматически проходит по выходному ребру к целевому узлу Написать письмо. Это инициирует выполнение действия, определенного узлом действия Написать письмо. По завершении действия Написать письмо поток маркера управления переходит в узел действия Адресовать письмо тогда и только тогда, когда удовлетворено его предусловие адрес известен. После выполнения постусловия письмо адресовано управление переходит от Адресовать письмо к Послать письмо. И наконец, поскольку условий, препятствующих выходу потока из Послать письмо, нет, поток управления прибывает в конечное состояние и деятельность завершается.

В этом простом примере поток управления проходит по всем узлам действия по очереди, обусловливая их выполнение. Это основная семантика деятельности.

Как уже упоминалось, состояние исполняющейся системы в любой момент времени может быть представлено расположением ее маркеров. Например, когда маркер находится в узле Написать письмо, можно сказать, что система находится в состоянии Написание письма. Однако не каждое выполнение действия или передача маркера создает заметное изменение в состоянии системы с точки зрения ее конечных автоматов (глава 21). Тем не менее расположение маркеров обеспечивает связь между диаграммами деятельности и диаграммами состояний, которые должны быть гарантированно согласованными для конкретного элемента модели.

Хотя семантика деятельностей UML 2 *описывается* «игрой» маркеров, они едва ли когда-нибудь реализуются таким образом. По сути, деятельность – это всего лишь описание, для которого возможно мно-

жество реализаций. Например, на рис. 14.5 описывается простой бизнес-процесс, а не программная система, и реализации этого процесса вообще *не* используют передачу маркеров!

14.6. Разделы деятельности

> Каждый раздел деятельности представляет высокоуровневую группировку взаимосвязанных действий.

Чтобы облегчить чтение диаграмм деятельности, можно разбить деятельности на разделы с помощью вертикальных, горизонтальных или кривых линий. Каждый раздел деятельности – это группа взаимосвязанных действий с высоким уровнем вложенности. Иногда разделы деятельности называют плавательными дорожками (swimlanes). Разбиение на разделы – мощный метод. При правильном использовании он может существенно упростить понимание диаграмм деятельности.

В UML 2 семантику разделов деятельности определяет разработчик модели. Разделы не имеют собственной семантики, поэтому могут использоваться для разбиения диаграмм деятельности любым удобным способом! Разделы деятельности обычно применяются для представления:

- прецедентов;
- классов;
- компонентов;
- организационных единиц (в бизнес-моделировании);
- ролей (в моделировании рабочих потоков).

Но этим не ограничиваются. Например, в проектных моделях распределенных систем разделы деятельности могут использоваться даже для моделирования распределения процессов на физических машинах.

У каждого множества разделов должно быть единственное измерение, описывающее его базовую семантику. В рамках этого измерения разделы могут быть иерархично вложенными. На рис. 14.6 показана деятельность, имеющая иерархично вложенное множество разделов деятельности.

Местонахождение – это измерение, и в рамках этого измерения существует иерархия разделов, как показано на рис. 14.7. Эта диаграмма моделирует бизнес-процесс создания курса нашей партнерской компании Zuhlke Engineering AG. Многие из их курсов разрабатываются нами в Лондоне.

Часто разделы деятельности и параллельные потоки управления взаимосвязаны. Моделирование параллелизма на диаграммах деятельности подробно рассматривается в разделе 14.8.3. Например, обычно разные подразделения или организационные единицы работают параллельно, затем в некоторый момент происходит их синхронизация.

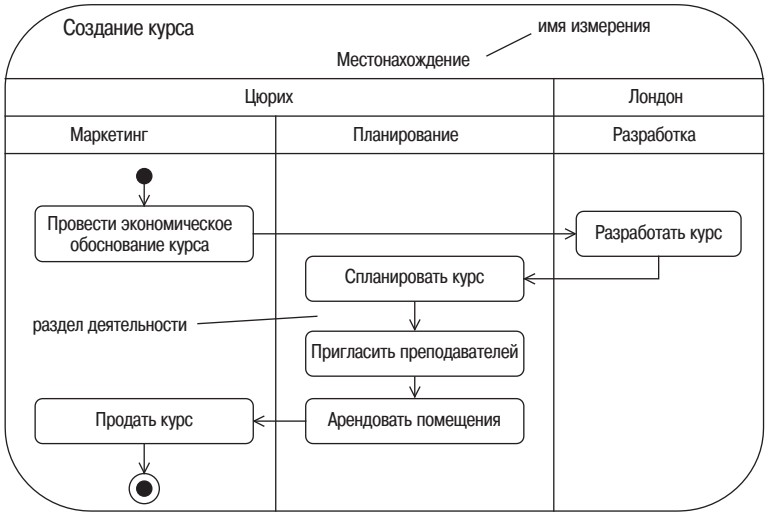


Рис. 14.6. *Деятельность с иерархично вложенным множеством разделов*

Диаграммы деятельности с разделами деятельности превосходно подходят для моделирования подобного процесса.

Иногда абсолютно нереально сгруппировать узлы в вертикальные или горизонтальные разделы, не ухудшив при этом читаемость диаграммы. В этом случае можно создавать разделы неправильной формы с помощью кривых или обозначать разделы с помощью текста. В UML существует текстовая нотация для разделов деятельности. Пример текстовой нотации приведен на рис. 14.8. Однако обычно к ней прибегают в самом крайнем случае, потому что графическое представление, как правило, намного нагляднее.

Местоположение действия в иерархии раздела можно задать с помощью разделенного двойными двоеточиями пути (pathname); путь указывается в скобках над именем действия. Если действие встречается в нескольких разделах, через запятую перечисляют имена путей каждого из разделов (рис. 14.8).

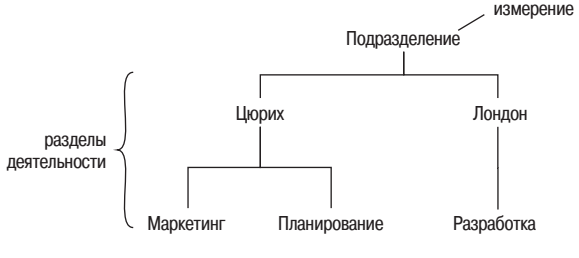


Рис. 14.7. *Иерархия разделов в рамках измерения*

Рис. 14.8. *Примеры текстовой нотации для разделов деятельности*

> Разделы, обозначенные стереотипом **«external»**, не являются частью системы.

Порой необходимо показать на диаграмме деятельности поведение, находящееся, строго говоря, вне области действия системы, например взаимодействие системы с некоторой внешней системой. Это можно представить, указав на диаграмме деятельности стереотип «external» прямо над именем раздела. Обратите внимание, что внешний раздел не является частью системы и, следовательно, не может входить в какую-либо иерархию разделов модели.

Тщательно выбирая измерения и разделы деятельности, можно, безусловно, добавить на диаграмму деятельности массу полезной информации. Однако эти возможности могут и усложнить диаграмму, особенно при наличии нескольких измерений и сложной иерархии разделов! На практике необходимо стремиться к использованию на одной диаграмме не более трех уровней иерархии (включая измерение) и не более двух измерений.

Всегда опирайтесь на собственное мнение и применяйте разделы деятельности только в случае, если они действительно повышают ценность модели.

14.7. Узлы действия

Узлы действия (action node) исполняются в следующих случаях:

- маркеры одновременно поступили на все входящие ребра
- и входящие маркеры удовлетворяют всем локальным предусловиям узла действия.

Это проиллюстрировано на рис. 14.9.

Узлы действия осуществляют операцию логическое И над своими входящими маркерами – узел не готов к исполнению до тех пор, пока маркеры не будут присутствовать на *всех* входящих ребрах. Даже если все необходимые маркеры присутствуют, узел будет исполняться только после удовлетворения его локального предусловия.

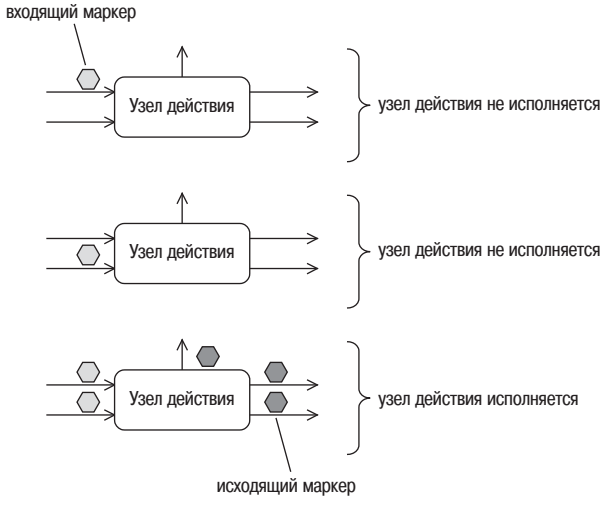


***Рис. 14.9.** Условия исполнения узлов действия*

> Узлы действия предлагают управляющие маркеры на *всех* своих исходящих ребрах – неявное ветвление.

По завершении выполнения узла действия проверяется локальное постусловие. Если оно удовлетворено, узел одновременно предлагает маркеры на *всех* своих исходящих ребрах. Это ветвление является неявным, поскольку один узел действия может породить множество потоков. В отличие от обычных блок-схем, диаграммы деятельности по сути своей параллельны.

Поскольку в узлах действия производятся некоторые действия, обычно их имена являются глаголами или глагольными группами. Спецификация UML не дает никаких рекомендаций по присваиванию имен узлам действия. Мы пользуемся соглашением, по которому имя узла начинается с большой буквы, все остальные слова, входящие в имя, пишутся с маленькой буквы через пробелы. Единственное исключение из этого правила: узел действия содержит ссылку на другой элемент модели. В этом случае имя элемента модели всегда используется как есть, *без* изменения регистра или добавления пробелов. На рис. 14.10 показано два примера. В верхнем примере происходит ссылка на что-то

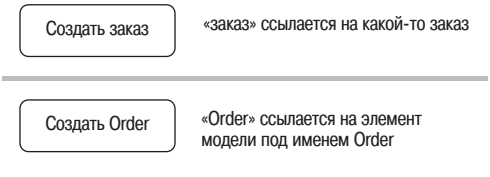


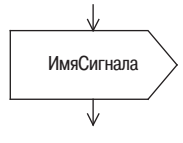
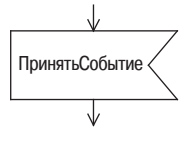
***Рис. 14.10.** Примеры имен узлов действия*

под названием «заказ», тогда как нижний пример явно ссылается на класс `Order`, который можно найти где-то в модели.

Детали действия фиксируются в описании узла действия. Часто это обычное текстовое описание, такое как «Написать письмо», но на этапе проектирования оно превращается в структурированный текст, псевдокод или реальный код. При моделировании прецедента диаграмма деятельности могла бы опережать поток прецедента на два-три шага. Но это может вызвать трудности, поскольку придется постоянно синхронизировать прецедент и соответствующую диаграмму деятельности.

Существует четыре типа узлов действия; они перечислены в табл. 14.1. Подробное обсуждение типов узлов действия можно найти в разделах, указанных в таблице.

Таблица 14.1

Синтаксис	Имя	Семантика	Раздел
	Узел вызова действия	Инициирует деятельность, поведение или операцию.	14.7.1
	Посылка сигнала	Действие посылки сигнала – посылает сигнал асинхронно (отправитель *не* ожидает подтверждения получения сигнала). Для создания сигнала может принимать входные параметры.	15.6
	Узел действия, принимающий событие	Принимает событие – ожидает события, установленного объектом-владельцем, и выдает событие на выходе. Активируется при получении маркера по входящему ребру. Если *нет* входящего ребра, запускается при запуске включающей его деятельности и всегда является активированным.	15.6
	Узел действия, принимающий событие времени	Принимает событие времени – отвечает на определенное значение времени. Генерирует события времени соответственно своему временному выражению.	14.7.2

14.7.1. Узел вызова действия

Самыми распространенными узлами действий являются *узлы вызова действия* (*call action node*). Этот тип узлов может инициировать:

- деятельность;
- поведение;
- операцию.

> Узел вызова действия может инициировать деятельность, поведение или операцию.

Некоторые примеры синтаксиса узла вызова действия приведены на рис. 14.11. Как видно из рисунка, синтаксис очень гибок!

- Посредством специального символа «грабли» в нижнем правом углу пиктограммы узла можно указать, что действие вызывает другое действие. Имя узла соответствует имени вызываемой им деятельности.
- Можно вызвать поведение – это прямой вызов поведения контекста деятельности *без* указания какой-либо конкретной операции.
- Можно вызвать операцию, используя стандартный синтаксис операции, описанный в разделе 7.5.3.
- Можно вызвать операцию, описывая ее детали на конкретном языке программирования. Это может быть особенно полезным при использовании инструментального средства UML, позволяющего генерировать код из диаграмм деятельности (например, iUML от компании Kennedy Carter, *www.kc.com*).
- С помощью ключевого слова self можно использовать возможности контекста деятельности.

Узлы вызова действия, используемые в диаграммах деятельности на уровне анализа, обычно вызывают поведение. Узлы вызова действия, инициирующие конкретные операции, как правило, используются для более детального моделирования деятельности при проектировании.

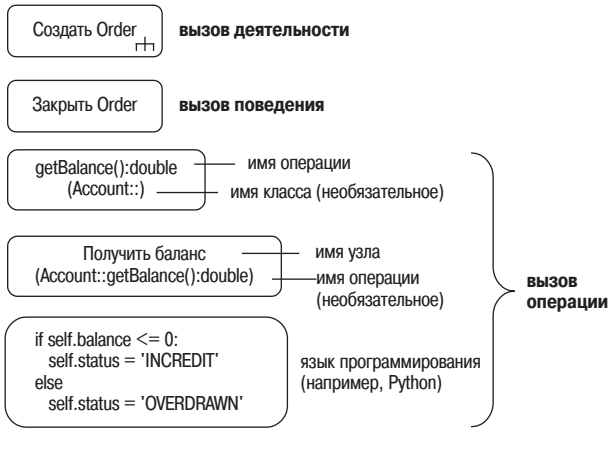


Рис. 14.11. *Примеры синтаксиса узла вызова действия*

14.7.2. Узел действия, принимающий событие времени

> Узел действия, принимающий события времени, реагирует на время.

Узел действия, принимающий события времени, реагирует на время. Этот тип узла имеет временное выражение и генерирует событие времени, когда это выражение становится истинным. Поведение такого узла зависит от наличия входящего ребра.

Например, на рис. 14.12 показан узел действия, принимающий события времени, без входящего ребра. Этот узел станет активным и будет генерировать событие времени после запуска его деятельности-владельца, когда его временное выражение становится истинным. В приведенном примере событие времени генерируется в конце каждого финансового года и инициирует деятельность Отправить налоговую декларацию компании.

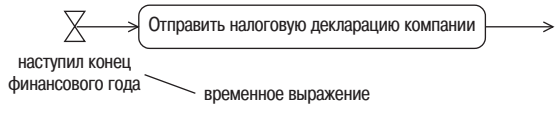


***Рис. 14.12.** Узел действия, принимающий событие времени, без входящего ребра*

Однако в примере на рис. 14.13 действие, принимающее событие времени, имеет входящее ребро и станет активным только после получения маркера по этому ребру. Этот пример является фрагментом системы управления лифтом. Первое действие открывает дверь лифта и запускает действие, принимающее событие времени. Это действие ожидает десять секунд и затем передает маркер действию Закрыть дверь.

***Рис. 14.13.** Узел действия, принимающий событие времени, с входящим ребром*

Обратите внимание, что временное выражение может указывать на:

- некоторое событие (например, конец финансового года);
- конкретный момент времени (например, 11/03/1960);
- временной интервал (например, ожидать 10 секунд).

14.8. Узлы управления

Узлы управления контролируют поток управления деятельности. В табл. 14.2 представлены все узлы управления UML 2; их подробное обсуждение см. в следующих разделах.

Таблица 14.2

Синтаксис	Имя	Семантика		Раздел
	Начальный узел	Указывает, где начинается поток при вызове деятельности.		14.8.1
	Конечный узел деятельности	Завершает деятельность.	Конечные узлы	14.8.1
	Конечный узел потока	Завершает определенный поток деятельности – другие потоки не затрагиваются.		14.8.1
	Узел решения	Поток проходит по исходящему ребру, сторожевое условие которого истинно. Может иметь входные данные (необязательно).		14.8.2
	Узел слияния	Копирует входные маркеры в единственное выходное ребро.		14.8.2
	Узел ветвления	Разделяет поток на несколько параллельных потоков.		14.8.3
	Узел объединения	Синхронизирует несколько параллельных потоков. Может иметь описание объединения (не обязательно) для изменения его семантики.		14.8.3

14.8.1. Начальный и конечный узлы

> Начальный узел показывает, где начинается деятельность.

Как уже говорилось в разделе 14.4, начальный узел (initial node) – это точка, в которой начинается поток при вызове деятельности. У деятельности может быть более одного начального узла. В этом случае потоки запускаются во всех начальных узлах одновременно и выполняются параллельно.

> Конечный узел деятельности завершает все потоки деятельности.

Деятельность также может быть инициирована действием принятия события (раздел 15.6) или узлом, являющимся параметром (раздел 14.9.3). Таким образом, начальные узлы *не* являются обязательными, поскольку есть другие способы запуска деятельности.

> Конечный узел потока завершает один из потоков деятельности.

Конечный узел (final node) деятельности завершает *все* потоки деятельности. Конечных узлов деятельности может быть много, и тот, который будет активирован первым, завершит все остальные потоки и саму деятельность.

Конечный узел потока просто останавливает *один* из потоков деятельности, остальные потоки продолжают выполнение. Пример приведен на рис. 15.10.

14.8.2. Узлы решения и слияния

Узел решения имеет одно входящее ребро и два и более альтернативных исходящих ребер. Маркер, поступающий по входящему ребру, будет предложен *всем* исходящим ребрам, но пройдет только по *одному* из них. Узел решения – это перекресток потоков, на котором маркер должен выбрать только один путь.

> Узел решения передает маркер на то выходное ребро, для которого выполняется сторожевое условие.

Каждое выходное ребро защищено *сторожевым условием* (*guard condition*), которое означает, что ребро примет маркер только в случае выполнения сторожевого условия. Важно, чтобы сторожевые условия были гарантированно взаимоисключающими, т. е. чтобы в любой момент времени истинным могло быть только *одно* из них. В противном случае согласно спецификации UML 2 поведение узла решения формально является неопределенным!

Для задания ребра, по которому пройдет поток управления в случае невыполнения всех сторожевых условий, может использоваться ключевое слово else.

На рис. 14.14 показан простой пример узла решения. После действия Получить корреспонденцию поток управления попадает в узел решения. Если выполняется условие [это мусор], почта отправляется в мусорную корзину, в противном случае (else) почтовое сообщение открывается.

Узел, отмеченный стереотипом «decisionInput» (входные данные решения), представляет условие принятия решения. Его результат используется сторожевыми условиями на исходящих ребрах. Пример фрагмента деятельности показан на рис. 14.15. Здесь условие принятия решения сравнивает запрашиваемую для снятия сумму с балансом счета. Если баланс больше или равен запрашиваемой сумме, условие принимает значение истина и поток переходит к действию Снять сумму. В противном случае регистрируется неплатежеспособность.

На рис. 14.14 показан узел слияния (merge node). В узлах слияния сходятся два или более входящих ребра и выходит одно исходящее. Они объединяют все входящие потоки в один исходящий. Семантика слияния очень проста: все маркеры, предлагаемые на входящих ребрах, предлагаются на исходящем ребре. Маркеры и поток не изменяются.

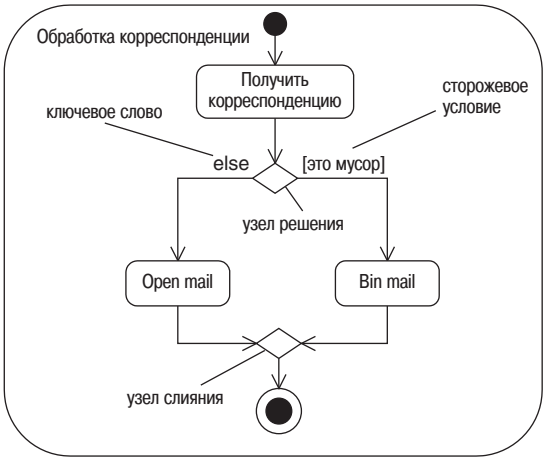


Рис. 14.14. *Пример узла решения и узла слияния*

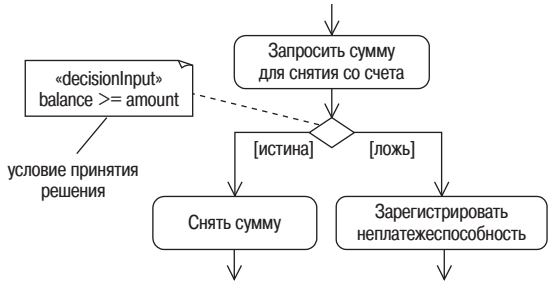


Рис. 14.15. *Фрагмент деятельности с узлом, помеченным стереотипом «decisioninput»*

Узел слияния и непосредственно следующий за ним узел решения могут быть объединены в один символ, как показано на рис. 14.16. Однако мы не рекомендуем применять такую сокращенную нотацию, поскольку изображение узлов по отдельности придает диаграмме большую наглядность.

14.8.3. Узлы ветвления и объединения – параллелизм

> Узел ветвления разделяет поток на несколько параллельных потоков.

Параллельные потоки деятельности можно создать путем разделения одного потока с помощью узла ветвления. Хотя обычно параллелизм является решением, принимаемым во время проектирования, нередко необходимо показать параллельные деятельности при моделировании бизнес-процессов. По этой причине мы будем часто использовать узлы ветвления и объединения как при анализе, так и при проектировании.

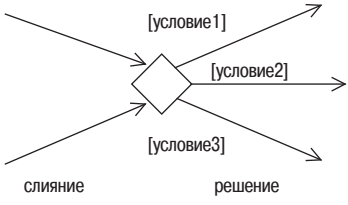


***Рис. 14.16.** Узел слияния и узел решения могут быть объединены в один символ*

Узел ветвления имеет одно входящее и два или более исходящих ребер. Маркеры, поступающие по входящим ребрам, дублируются и предлагаются на *всех* исходящих ребрах одновременно. Тем самым единственный входящий поток разделяется на несколько параллельных исходящих потоков. У каждого исходящего ребра может быть сторожевое условие, и маркер, как и в узлах решения, может передаваться по исходящему ребру только в случае выполнения сторожевого условия.

> Узел объединения синхронизирует и объединяет несколько входящих потоков в единственный исходящий.

В узле объединения несколько входящих ребер встречаются и объединяются в одно исходящее. Эти узлы синхронизируют потоки: маркер на их единственном исходящем ребре предлагается только после того, как поступили маркеры *всех* входящих потоков. Они осуществляют операцию логического И над всеми своими входящими ребрами.

На рис. 14.17 показан простой пример Процесс производства продукта, в котором используются узлы ветвления и объединения. В этом примере:

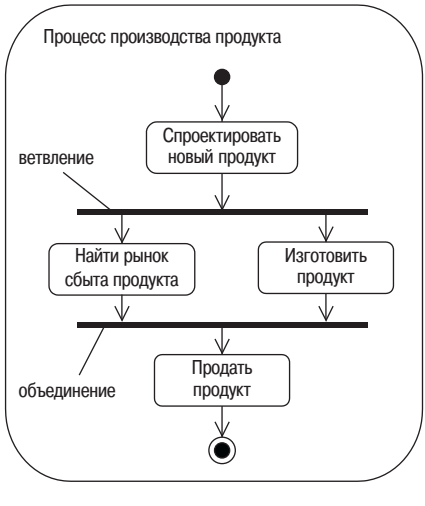


***Рис. 14.17.** Деятельность Процесс производства продукта включает **у**злы ветвления и объединения*

- продукт сначала разрабатывается;
- поиск рынка сбыта и изготовление продукта осуществляются параллельно;
- продукт реализуется *только* после завершения обоих процессов – поиска рынка сбыта и изготовления.

На рис. 14.17 деятельность Процесс производства продукта начинается с действия Спроектировать новый продукт. После этого узел ветвления разделяет единый поток на два параллельных. В одном из этих потоков ведется поиск рынка сбыта продукта (Найти рынок сбыта продукта), в другом – продукт изготавливается (Изготовить продукт). Узел объединения синхронизирует эти два параллельных потока, поскольку ожидает маркер от каждого из параллельных действий. Получив маркер от каждого действия, он предлагает маркер на своем выходном ребре, и поток переходит к действию Продать продукт.

При моделировании узлов объединения важно гарантировать получение маркера всеми входными ребрами. Например, на рис. 14.17 узел объединения никогда не смог бы получить подходящие маркеры для активации, если бы на исходящие потоки ветвления были наложены взаимоисключающие сторожевые условия. Это привело бы к «зависанию» деятельности.

14.9. Объектные узлы

> Объектные узлы показывают, что экземпляры классификатора доступны.

Объектные узлы – это специальные узлы, показывающие, что экземпляры конкретного классификатора доступны в данной точке деятельности. Они обозначены именем классификатора и представляют его экземпляры или подклассы. Фрагмент деятельности на рис. 14.18 показывает объектный узел, представляющий экземпляры классификатора Order или подклассы Order.

> Потоки объектов представляют движение объектов в деятельности.

Входящие и исходящие ребра объектных узлов называют *потоками объектов* (*object flows*). Это особые типы потоков, представляющие

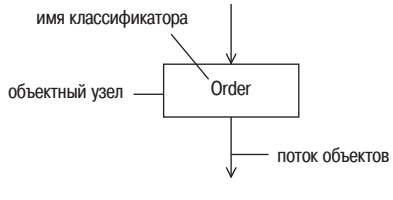


***Рис. 14.18.** Объектный узел*

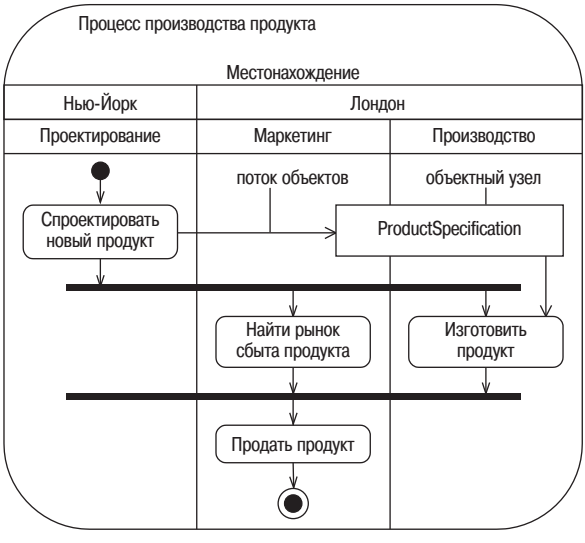


Рис. 14.19. *Деятельность Процесс производства продукта дополнена разделами, действием Спроектировать новый продукт и объектом ProductSpecification*

движение объектов в деятельности. Сами объекты создаются и используются узлами действия.

На рис. 14.19 показана деятельность Процесс производства продукта, впервые представленная на рис. 14.17. Она была дополнена: включены разделы и действием Спроектировать новый продукт создается объект ProductSpecification (спецификация продукта), который используется действием Изготовить продукт для описания производственного процесса.

> Выходные ребра объектного узла конкурируют за каждый выходной маркер.

Когда объектный узел получает объектный маркер по одному из своих входных ребер, он предлагает его всем выходным ребрам одновременно, и эти ребра *конкурируют* за этот маркер. Главное то, что маркер всего один – он *не* тиражируется на все ребра! Этот маркер получает поток, готовый первым принять его.

14.9.1. Семантика буфера объектного узла

Объектные узлы имеют очень интересную семантику. Они действуют как буферы – участки деятельности, где могут находиться объектные маркеры в ожидании принятия другими узлами.

> Объектные узлы выступают в роли буферов.

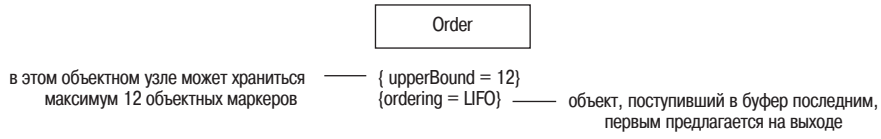


Рис. 14.20. *Объектный узел с верхней границей*

По умолчанию каждый объектный узел может удерживать бесконечное число объектных маркеров. Однако иногда необходимо ограничить размер буфера. Для этого задают *верхнюю границу* (*upper bound*) объектного узла. Она показывает максимальное число маркеров, которые могут удерживаться в узле в любой момент времени. Узел принимает объектные маркеры до тех пор, пока не заполнится. Пример объектного узла с заданной верхней границей приведен на рис. 14.20.

Для объектных узлов можно задать два аспекта семантики буфера.

- У объектных узлов есть *порядок расположения* (*ordering*) (рис. 14.20), определяющий поведение буфера. Применяемым по умолчанию порядком является FIFO (first-in, first-out – первым вошел, первым вышел). Это означает, что объект, первым поступивший в буфер, первым предлагается его выходным ребрам. Существует обратный порядок расположения – LIFO (last-in, first-out – последним вошел, первым вышел).
- Объектные узлы могут обладать *селективным поведением* (*selection behavior*). Это закрепленное за узлом поведение, по которому объекты из входных потоков выбираются согласно некоторому критерию, определенному разработчиком модели. Критерий задается примечанием со стереотипом «selection» (выбор), как показано на рис. 14.21. В данном примере объектный узел выбирает только те объекты Order, которые были созданы в декабре, и предлагает их своим выходным потокам в применяемом по умолчанию порядке (FIFO).

Объектный узел может использоваться для сбора объектов из нескольких входящих объектных потоков или для распределения объектов по нескольким исходящим объектным потокам. В этих случаях узел используется исключительно из-за его буферной семантики. Таким образом, чтобы подчеркнуть этот факт, узел можно обозначить стереотипом «centralBuffer» (центральный буфер).

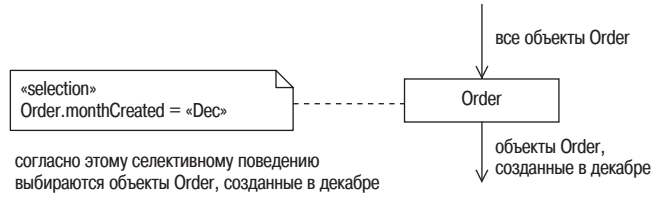


Рис. 14.21. *Объектный узел с селективным поведением*

Наряду с отдельными объектами объектные узлы могут буферизовать *множества* (*sets*) объектов. Множество – это коллекция объектов, в которой нет дублирования, т. е. каждый объект имеет уникальный идентификатор. Чтобы показать это, перед именем классификатора просто указывают Set of (множество). Пример приведен на рис. 14.23.

Немного подробнее стереотипы «selection» и «centralBuffer» рассматриваются в разделах 15.8.2 и 15.11.

14.9.2. Представление объектов в состоянии

> Объектные узлы могут представлять объекты, находящиеся в определенном состоянии.

Объектные узлы могут представлять объекты, находящиеся в определенном состоянии. Например, на рис. 14.22 показан фрагмент деятельности обработки заказа, которая принимает объекты Order, находящиеся в состоянии Открытый, и отправляет их по назначению. Состояния объектов, на которые ссылаются объектные узлы, могут быть смоделированы с помощью конечных автоматов (см. главу 21).

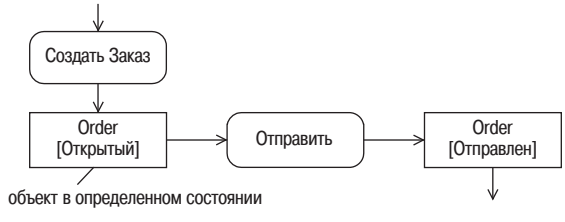


***Рис. 14.22.** Объект Order находится в состоянии Открытый*

14.9.3. Параметры деятельности

> Параметры деятельности – это объектные узлы, поступающие в или исходящие из деятельности.

Объектные узлы могут использоваться для обеспечения входных и выходных данных деятельности, как показано на рис. 14.23. Входящие и исходящие объектные узлы должны перекрывать рамку деятельности. Входящие объектные узлы связаны с деятельностью одним или более исходящими ребрами, а у *исходящих* объектных узлов – одно или более *входящих* ребер, поступающих из деятельности.

На рис. 14.23 деятельность Производство продукта на заказ имеет три входных параметра: CustomerRequest (запрос клиента), Set of BusinessConstraint (множество бизнес-ограничений) и Order, а также один выходной параметр, Order. Узел Set of BusinessConstraint содержит множество объектов BusinessConstraint.

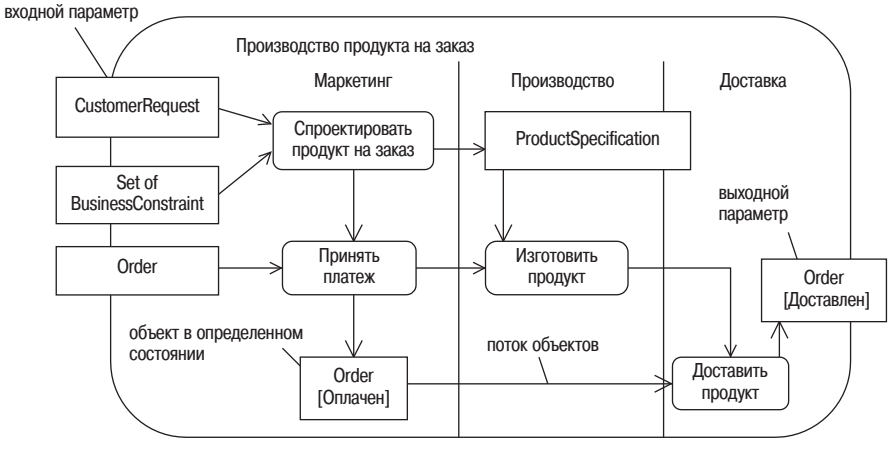


Рис. 14.23. *Деятельность Производство продукта на заказ имеет три входных параметра и один выходной*

К этому процессу предъявляется несколько бизнес-требований:

- Продукты проектируются на основании CustomerRequest. При этом создается ProductSpecification (спецификация продукта).
- При проектировании продукта учитываются все BusinessConstraint.
- Оплата производится только после завершения проектирования продукта.
- Изготовление продукта не может быть начато до тех пор, пока не будет получен платеж и создана спецификация продукта (объект ProductSpecification).
- Доставка не может осуществляться до тех пор, пока продукт не будет изготовлен.

Проанализируем данную деятельность.

1. Деятельность начинается, когда по входным потокам объектов действия Спроектировать продукт на заказ поступают CustomerRequest и множество BusinessConstraint. Действие принимает входящие объекты и выдает объект ProductSpecification.
2. Действие Принять платеж выполняется, когда получает управляющий маркер от Спроектировать продукт на заказ и объект Order по входному потоку объектов. Оно меняет состояние объекта Order на Оплачен и выдает его на свой единственный выходной поток объектов.
3. Затем поток управления переходит к действию Изготовить продукт. Оно принимает объект ProductSpecification, производимый в действии Спроектировать продукт на заказ, и предлагает маркер управления действию Доставить продукт.
4. Доставить продукт выполняется, когда от Изготовить продукт поступает маркер управления и объект Order находится в состоянии Оплачен.

Результатом действия будет объект Order в состоянии Доставлен. Этот объект Order является выходным параметром деятельности.

Как видите, мы довольно легко смогли выполнить бизнес-требования.

- Мы не будем пытаться Спроектировать продукт на заказ до тех пор, пока клиент не сделает запрос (CustomerRequest) и не будет получено множество бизнес-ограничений (Set of BusinessConstraint).
- Мы не можем Принять платеж, пока не будет сделан заказ (объект Order) и не будет завершено действие Спроектировать продукт на заказ.
- Мы не можем Изготовить продукт до тех пор, пока нет спецификации продукта (ProductSpecification) и не завершено действие Принять платеж (иначе говоря, пока продукт не оплачен!).
- Мы не можем Доставить продукт, пока он не изготовлен (не завершилось действие Изготовить продукт) и не оплачен (пока объект Order не перейдет в состояние Оплачен).

Этот пример иллюстрирует мощь диаграмм деятельности. Они могут кратко и точно моделировать сложные процессы.

14.10. Контакты

Деятельность, в которой много потоков объектов, может стать очень запутанной. Чтобы немного прояснить ситуацию, используйте контакты!

> Контакт – это объектный узел, представляющий один вход или выход из действия.

Контакт – это просто объектный узел, представляющий один вход или выход из действия. У входных контактов только одно входное ребро, а у выходных – только одно выходное ребро. Во всем остальном их семантика и синтаксис аналогичны объектным узлам. Однако из-за их небольшого размера всю информацию, например имя классификатора, приходится указывать вне контакта, но как можно ближе к нему.

На рис. 14.24 показана деятельность Войти в систему, имеющая два потока объектов. Деятельность начинается с действия Получить UserName. Его результатом является допустимый (valid) объект UserName. Следующее действие – Получить Password, на выходе которого получается допустимый объект Password. Деятельность Аутентифицировать объект User начинает выполнение, когда получает действительные объекты UserName и Password по своим входным потокам объектов. Осуществляется аутентификация пользователя, и деятельность завершается.

На рис. 14.25 показана та же деятельность, но представленная с использованием контактов. Как видите, нотация контактов выглядит более компактно, и диаграмма становится более аккуратной.

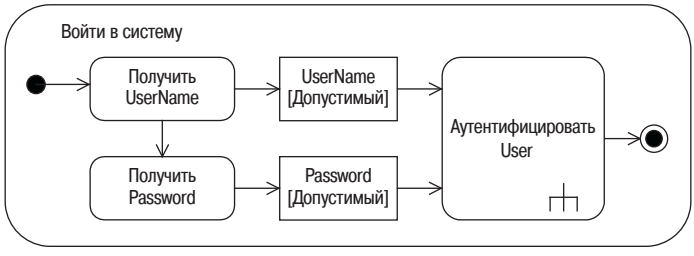


Рис. 14.24. *Деятельность с двумя потоками объектов*

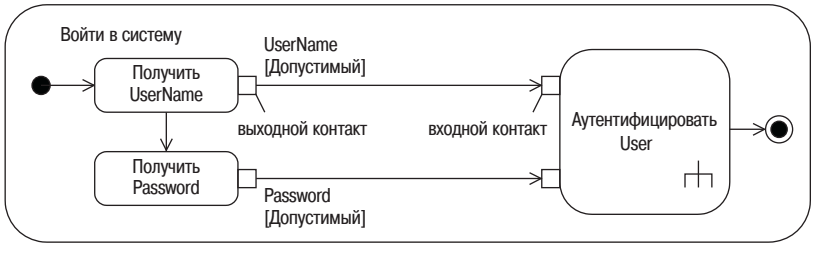


Рис. 14.25. *Та же деятельность с применением контактов*

Сравнивая рис. 14.24 и рис. 14.25, можно заметить, что объектный узел UserName эквивалентен комбинации выходного контакта действия Получить UserName и входного контакта действия Аутентифицировать объект User. Поэтому объектные узлы иногда называют *автономным контактом* (*stand-alone style pin*).

14.11. Что мы узнали

В этой главе было показано, что диаграммы деятельности можно использовать для моделирования множества различных процессов. Мы узнали следующее:

- Диаграммы деятельности – это ОО блок-схемы:
  - они используются для моделирования всех типов процессов;
  - диаграммы деятельности можно создать для *любого* элемента модели для описания его поведения;
  - хорошая диаграмма деятельности описывает один конкретный аспект поведения системы;
  - в UML 2 диаграммы деятельности имеют семантику технологии сетей Петри.
- Деятельности – это схемы, состоящие из узлов, соединенных ребрами.
  - Категории узлов:
    - узлы действия – элементарные единицы работы в рамках деятельности;

    - узлы управления – управляют потоком деятельности;
    - объектные узлы – представляют объекты, используемые в деятельности.
  - Категории ребер:
    - потоки управления – представляют поток управления деятельности;
    - потоки объектов – представляют поток объектов деятельности.
  - Маркеры перемещаются по сети (network) и могут представлять:
    - поток управления;
    - объект;
    - некоторые данные.
  - Движение маркеров по ребрам от начального узла к целевому узлу зависит от:
    - постусловий начального узла;
    - сторожевых условий ребра;
    - предусловий целевого узла.
  - Деятельности могут иметь предусловия и постусловия.
- Узлы действия.
  - Выполняются при одновременном поступлении маркеров по всем входным ребрам И удовлетворении всех предусловий.
  - После выполнения узлы действия предлагают маркеры *одновременно* на всех выходных ребрах, постусловия которых удовлетворены:
    - неявное ветвление.
  - Узел вызова действия:
    - инициирует деятельность – используется символ «грабли»;
    - инициирует поведение;
    - инициирует операцию.
  - Узел действия, посылающий сигнал (см. раздел 15.6).
  - Узел действия, принимающий событие (см. раздел 15.6).
  - Узел действия, принимающий событие времени, выполняется, когда временное выражение становится истинным:
    - некоторое событие (например, конец финансового года);
    - конкретный момент времени (например, 11/03/1960);
    - временной интервал (например, ожидать 10 секунд).
- Узлы управления:
  - начальный узел показывает, где начинается поток при вызове деятельности;
  - конечный узел деятельности заканчивает деятельность;

    - конечный узел потока заканчивает конкретный поток деятельности;
    - узел решения – поток направляется по исходящему ребру, сторожевое условие которого истинно:
        - может иметь стереотип «decisionInput»;
    - узел слияния копирует входные маркеры в единственное исходящее ребро;
    - узел ветвления разделяет поток на несколько параллельных потоков;
    - узел объединения синхронизирует несколько параллельных потоков:
        - может иметь {описание объединения}.
- Разделы деятельности – высокоуровневая группировка взаимосвязанных действий.
    - Разделы формируют иерархию, корнем которой является измерение.
- Объектные узлы представляют экземпляры классификатора.
    - Входящие и исходящие ребра – потоки объектов – представляют движение объектов.
    - Исходящие ребра объектного узла конкурируют за каждый исходящий маркер.
    - Объектные узлы работают как буферы:
        - { upperBound = n }
        - { ordering = FIFO } XOR { ordering = LIFO };
        - по умолчанию применяется { ordering = FIFO };
        - могут иметь стереотип «selection».
    - Объектные узлы могут представлять объекты, находящиеся в определенном состоянии:
        - должны соответствовать конечным автоматам.
    - Параметры деятельности – это объектные узлы, входящие в или исходящие из деятельности:
        - на диаграмме перекрывают рамку деятельности;
        - входящие параметры имеют один или более исходящих ребер, поступающих *в* деятельность;
        - исходящие параметры имеют один или более входящих ребер, поступающих *из* деятельности.
- Контакт – это объектный узел, представляющий один вход в или выход из действия или деятельности.

15

Дополнительные аспекты диаграмм деятельности

15.1. План главы

В этой главе рассматриваются дополнительные возможности диаграмм деятельности. Они вряд ли будут использоваться каждый день, но могут быть очень полезны в определенных ситуациях моделирования. Разделы этой главы можно читать в любой последовательности. Можно вообще только просмотреть главу, чтобы получить общее представление, а затем обращаться к соответствующим разделам и в случае необходимости использовать ту или иную конкретную возможность.

15.2. Разъемы

Основное правило: применения разъемов необходимо избегать. Однако если диаграмма очень сложна и не поддается упрощению, разъемы можно использовать для разрыва длинных ребер, которые трудно проследить, и для «распутывания» пересекающихся ребер. Это может упростить диаграммы деятельности и повысить их удобочитаемость.

Синтаксис разъема представлен на рис. 15.2. В заданной деятельности каждому исходящему разъему *должен* соответствовать единственный входящий разъем с такой же меткой. Метки – это идентификаторы разъема, никакой другой семантики у них нет. Обычно в их качестве выступают буквы алфавита.

> Разъемы могут разрывать длинные ребра, которые трудно проследить, и «распутывать» пересекающиеся ребра.

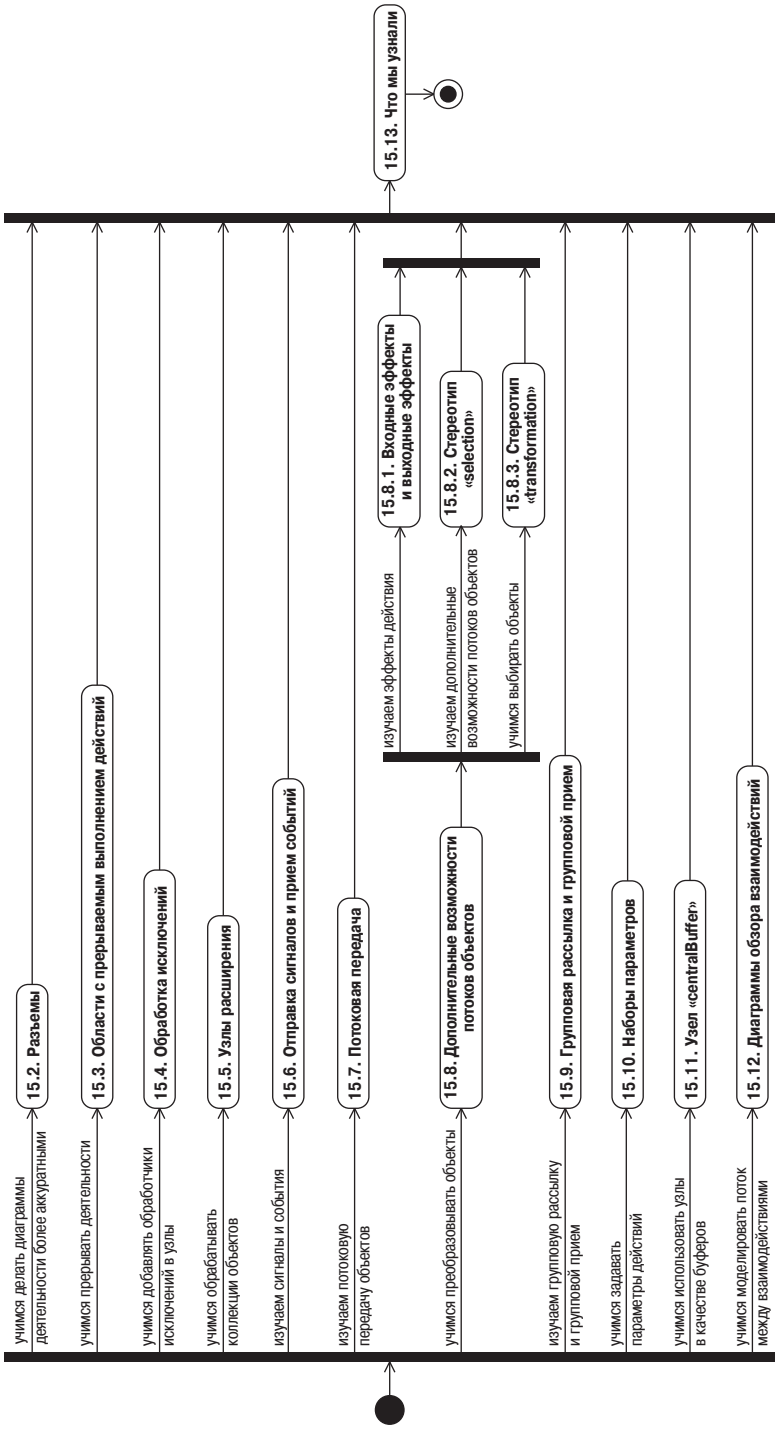


Рис. 15.1. План главы

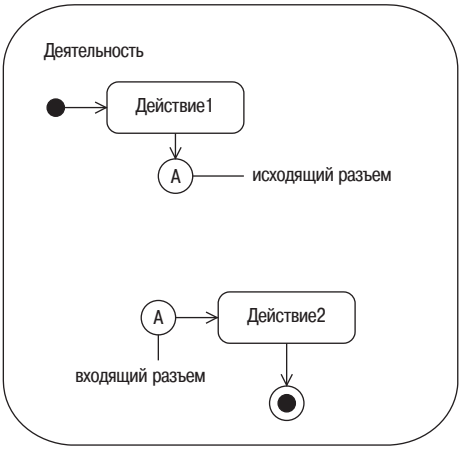


***Рис. 15.2.** Синтаксис разъема*

15.3. Области с прерываемым выполнением действий

Области с прерываемым выполнением действий – это области деятельности, прерываемые при прохождении маркера по прерывающему ребру. Когда область прерывается, все ее потоки немедленно прекращаются. Области с прерываемым выполнением действий полезны для моделирования прерываний и асинхронных событий. Чаще всего они используются при проектировании, но также могут быть полезны при анализе для представления процесса обработки асинхронных бизнес-событий.

> Области с прерываемым выполнением действий – это области деятельности, прерывающиеся при прохождении маркера по прерывающему ребру.

На рис. 15.3 показана простая деятельность Войти в систему, имеющая область с прерываемым выполнением действий. Область обозначена отрисованным пунктирной линией прямоугольником со скругленными углами, включающим действия Получить UserName, Получить Password и Отменить. Если принимающее событие действие Отменить получает событие Cancel в тот момент, когда управление находится в этой области, оно выводит маркер на прерывающее ребро и внезапно прекращает область. Все действия – Получить UserName, Получить Password и Отменить – прерываются.

Прерывающие ребра изображаются как зигзагообразные стрелки (рис. 15.3) или как обычные стрелки с пиктограммой зигзага над ними (рис. 15.4).

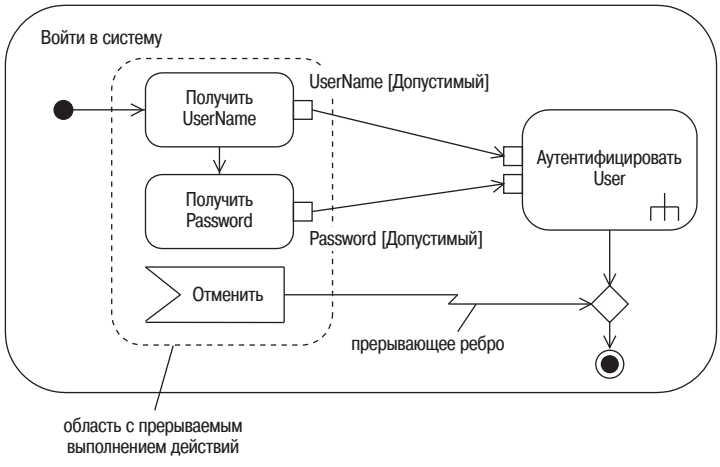


Рис. 15.3. *Деятельность Войти в систему имеет область с прерываемым выполнением действий*

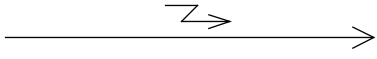


Рис. 15.4. *Вариант отображения прерывающего ребра*

15.4. Обработка исключений

Современные языки программирования часто обрабатывают ошибки посредством механизма, называемого *обработкой исключений* (*exception handling*). В случае выявления ошибки в защищенной части кода создается объект исключения. Поток управления переходит в обработчик исключения, который некоторым образом обрабатывает объект исключения. В этом объекте исключения содержится информация об ошибке, которая может использоваться обработчиком исключения. Обработчик исключения может прервать приложение или попытаться восстановить нормальное состояние. Часто информация объекта исключения сохраняется в журнале регистрации ошибок.

Обработку исключений на диаграммах деятельности можно моделировать с помощью контактов исключений, защищенных узлов и обработчиков исключений.

> У защищенного узла есть обработчик исключений.

На рис. 15.5 представлена обновленная деятельность Войти в систему. Теперь деятельность Аутентифицировать User выдает объект LogOnException (исключение при входе) при неудачной аутентификации пользователя. Этот объект принимается действием Зарегистрировать ошибку, ко-

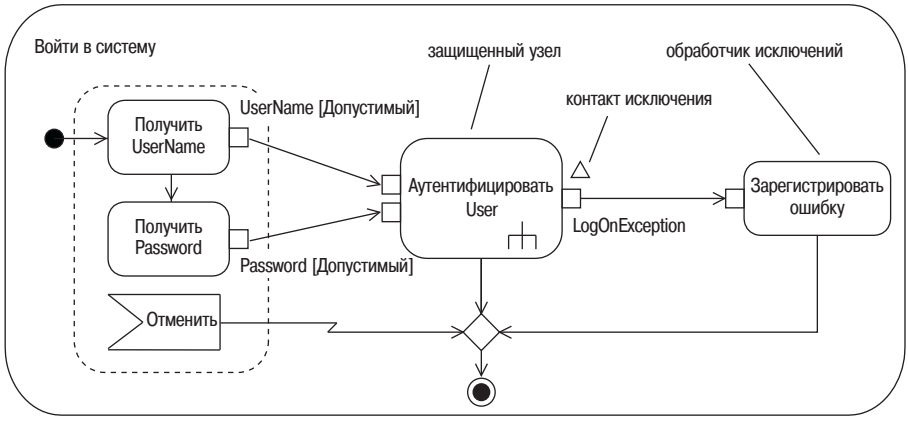


Рис. 15.5. *Деятельность Войти в систему с обработчиком исключений*

торое записывает информацию об ошибке в журнал регистрации ошибок. Для того чтобы показать, что выходной контакт представляет собой объект исключения, его помечают небольшим равносторонним треугольником (рис. 15.5). Узел Зарегистрировать ошибку выступает в роли обработчика исключений, генерируемых действием Аутентифицировать User. Если с узлом ассоциирован обработчик исключений, узел называют защищенным.

Поскольку обработка исключений обычно является задачей проектирования, а не анализа, защищенные узлы чаще используются при проектировании. Однако иногда может быть полезным смоделировать защищенный узел уже на стадии анализа, если он имеет важную бизнес-семантику.

15.5. Узлы расширения

Узлы расширения позволяют показать обработку коллекции объектов как часть диаграммы деятельности, называемую *областью расширения* (*expansion region*). Представление процесса обработки коллекции может быть довольно сложным и пространным, поэтому данная техника очень полезна как при анализе, так и при проектировании.

> Узел расширения – коллекция объектов, входящая в или выходящая из области расширения, исполняемой один раз для каждого объекта.

Узел расширения – это объектный узел, который представляет коллекцию объектов, входящую в или выходящую из области расширения. Область расширения исполняется по одному разу для каждого входного элемента. На рис. 15.6 показан пример области расширения. Она представлена в виде отрисованного пунктиром прямоугольника со скруг-

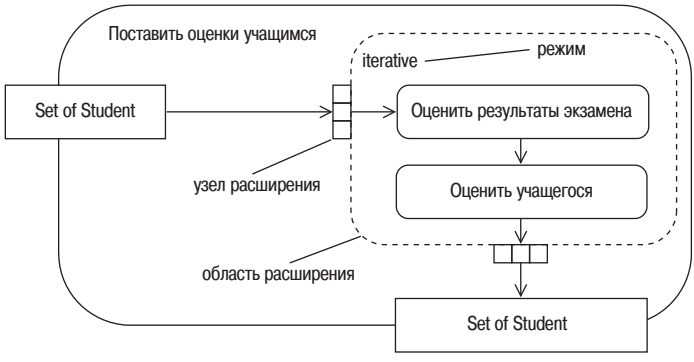


Рис. 15.6. *Пример области расширения*

ленными углами и входящим и исходящим узлами расширения. Узел расширения выглядит как контакт, но с тремя квадратными ячейками. Это указывает на поступление коллекции, а не одного объекта.

На узлы расширения накладываются два ограничения.

- Тип выходной коллекции *должен* соответствовать типу входной коллекции.
- Типы объектов входной и выходной коллекций *должны* быть одинаковыми.

Эти ограничения означают, что области расширения не могут использоваться для преобразования входных объектов одного типа в выходные объекты другого типа.

Количество выходных коллекций может не совпадать с количеством входных, поэтому области расширения могут использоваться для объединения или разделения коллекций.

У каждой области расширения есть режим, определяющий порядок обработки элементов входной коллекции. Режим может быть следующим:

- iterative (итеративный) – каждый элемент входной коллекции обрабатывается последовательно;
- parallel (параллельный) – элементы входной коллекции обрабатываются параллельно;
- stream (потоковый) – элементы входной коллекции обрабатываются в порядке поступления в узел.

Режим всегда должен быть задан явно, потому что спецификация UML не определяет применяемого по умолчанию режима.

На рис. 15.6 область расширения принимает набор объектов Student (учащийся). Она обрабатывает каждый из этих объектов по очереди (режим = iterative) и выдает набор обработанных объектов Student. Два

действия внутри области сначала определяют результаты экзамена Student и затем ставят ему оценку. В данном случае выходная коллекция объектов Student предлагается на выходном узле расширения только после того, как обработаны *все* объекты Student. Однако если бы был задан режим stream, объекты Student предлагались бы на выходном узле расширения сразу после обработки.

15.6. Отправка сигналов и прием событий

Сигнал представляет асинхронно передаваемую между объектами информацию. Сигнал моделируется как класс, отмеченный стереотипом «signal» (сигнал). Передаваемая информация хранится в атрибутах сигнала. При анализе сигналы могут использоваться для отображения отправки и получения асинхронных бизнес-событий (таких как OrderReceived (заказ получен)), а при проектировании они могут иллюстрировать асинхронный обмен информацией между разными системами, подсистемами или частями оборудования.

> Сигналы представляют асинхронно передаваемую между объектами информацию.

На рис. 15.7 показаны два сигнала, которые используются в деятельности, представленной на рис. 15.8.

Оба этих сигнала имеют тип *SecurityEvent* (событие системы безопасности). Сигнал AuthorizationRequestEvent (событие запроса авторизации) передает PIN и информацию карты, вероятно, в зашифрованном виде. В сигнале AuthorizationEvent (событие авторизации) хранится логический флаг, указывающий, были ли авторизованы карта и PIN.

> Узел действия, отправляющий сигнал, представляет отправку сигнала.

Сигнал можно послать с помощью узла действия, отвечающего за отправку сигнала. Он посылает сигнал асинхронно – деятельность, от-

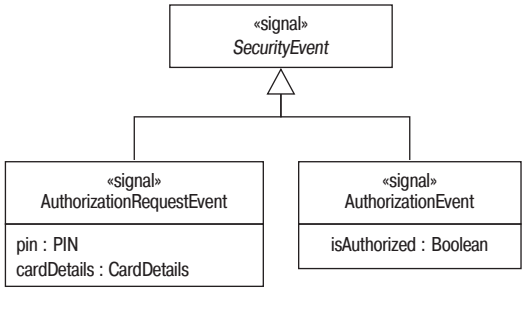


***Рис. 15.7.** Два сигнала, используемые в деятельности Проверить кредитную карту*

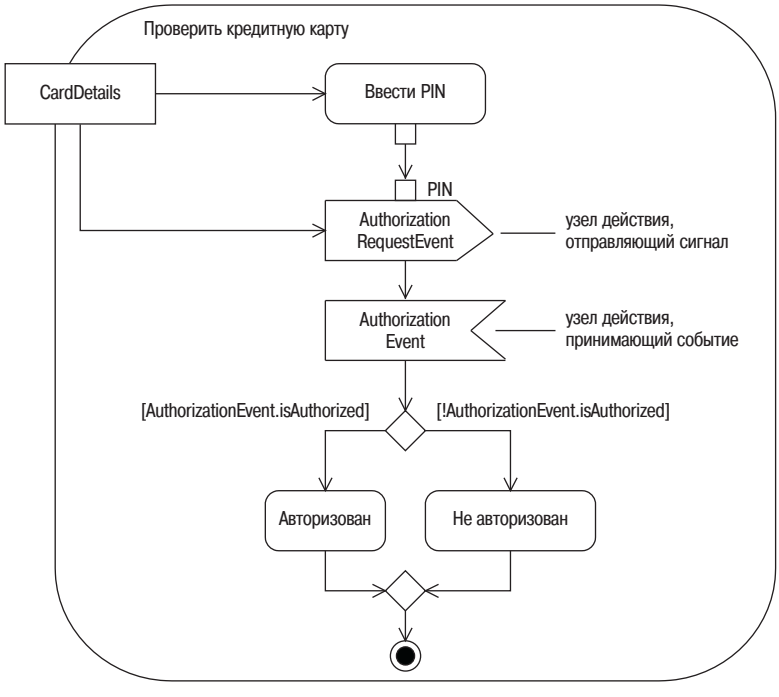


Рис. 15.8. *Деятельность Проверить кредитную карту*

правляющая сигнал, *не* ожидает подтверждения получения сигнала. Посылающая сигнал деятельность имеет следующую семантику:

- Посылающая сигнал деятельность инициируется тогда, когда маркер одновременно присутствует на *всех* ее входных ребрах. Если у сигнала имеются входные контакты, он должен получить входной объект соответствующего типа для каждого из своих атрибутов.
- При выполнении действия создается и посылается объект сигнала. Целевой объект обычно не указывается, но в случае необходимости его можно передать на входной контакт действия, отправляющего сигнал.
- Действие отправки не ожидает подтверждения принятия сигнала – оно асинхронно.
- Действие завершается, и маркеры управления предлагаются на его выходных ребрах.

> Узел действия, принимающий событие, ожидает получения события соответствующего типа.

Узел действия, принимающий событие, не имеет ни одного или имеет одно входное ребро. Он ожидает асинхронных событий, выявленных

его контекстом-владельцем, и предлагает их на своем единственном выходном ребре. Семантика его следующая.

- Действие приема события запускается входящим ребром управления; если входящих ребер нет, оно запускается при вызове деятельности-владельца.
- Действие ожидает получения события определенного типа. Это событие называют *триггером* (*trigger*).
- Когда действие получает триггер события соответствующего типа, оно выдает маркер, описывающий событие. Если событие было сигналом, маркер является сигналом.
- Действие продолжает принимать события до тех пор, пока выполняется деятельность.

На рис. 15.8 показана деятельность Проверить кредитную карту, отправляющая события AuthorizationRequestEvents и принимающая события AuthorizationEvents.

Ниже приведен поэтапный анализ деятельности Проверить кредитную карту.

1. Деятельность Проверить кредитную карту начинается, когда получает входной параметр CardDetails. Затем она предлагает пользователю ввести PIN-код.
2. Действие AuthorizationRequestEvent начинает выполняться, как только на его входные ребра поступают объекты PIN и CardDetails (информация карты). Используя эти входные параметры, оно создает сигнал AuthorizationRequestEvent и посылает его. Действия отправки сигналов изображаются в виде выпуклых пятиугольников, как показано на рисунке.
3. Сигналы отправляются асинхронно, и поток управления незамедлительно переходит к действию приема события AuthorizationEvent, которое изображено в виде вогнутого пятиугольника. Это действие ожидает получения сигнала AuthorizationEvent.
4. Получив этот сигнал, поток переходит в узел принятия решения. Если AuthorizationEvent.isAuthorized истинно, вызывается действие Авторизован, в противном случае выполняется действие Не авторизован.

Рассмотрим другой пример действий, принимающих события. На рис. 15.9 моделируется деятельность Показать новости, имеющая два принимающих события действия, которые инициируются автоматически при запуске деятельности. Когда действие приема события NewsEvent получает событие NewsEvent (событие новостей), оно передается действию Отобразить новости. После этого поток управления переходит в конечный узел потока, и этот конкретный поток завершается. Однако выполнение деятельности продолжается, и оба действия приема событий продолжают ожидать события. Когда деятельность получает событие TerminateEvent (событие завершения), управление переходит в ко-

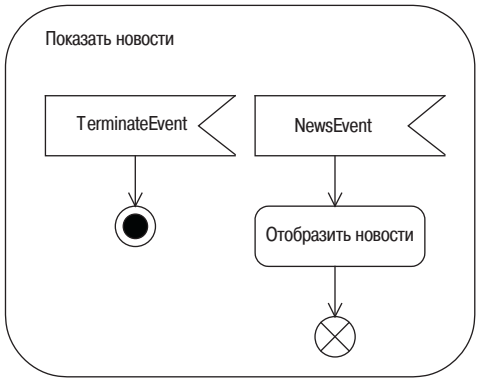


Рис. 15.9. *Деятельность Показать новости*

нечный узел деятельности, и вся деятельность, включая оба действия приема событий, немедленно завершается.

Если необходимо показать движение сигналов по диаграмме деятельности, их можно представить как объектные узлы. Хотя семантика сигналов аналогична семантике остальных объектных узлов, они имеют особый синтаксис (рис. 15.10). Их символ является сочетанием пиктограмм действия отправки сигнала и действия приема события, так что запомнить их легко.

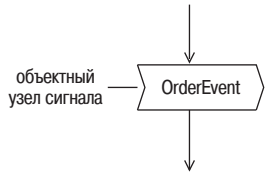


Рис. 15.10. *Синтаксис сигналов*

15.7. Потоковая передача

Обычно действия принимают маркеры со своих входных ребер только тогда, когда начинают выполнение, и предлагают их на своих выходных ребрах по завершении выполнения. Однако иногда требуется, чтобы действие выполнялось *непрерывно*, периодически принимая и предлагая маркеры. Такое поведение называют потоковой передачей. В UML 2 оно может быть проиллюстрировано любым из четырех способов, как показано на рис. 15.11.

> Потоковая передача – действие выполняется непрерывно, принимая и предлагая маркеры.

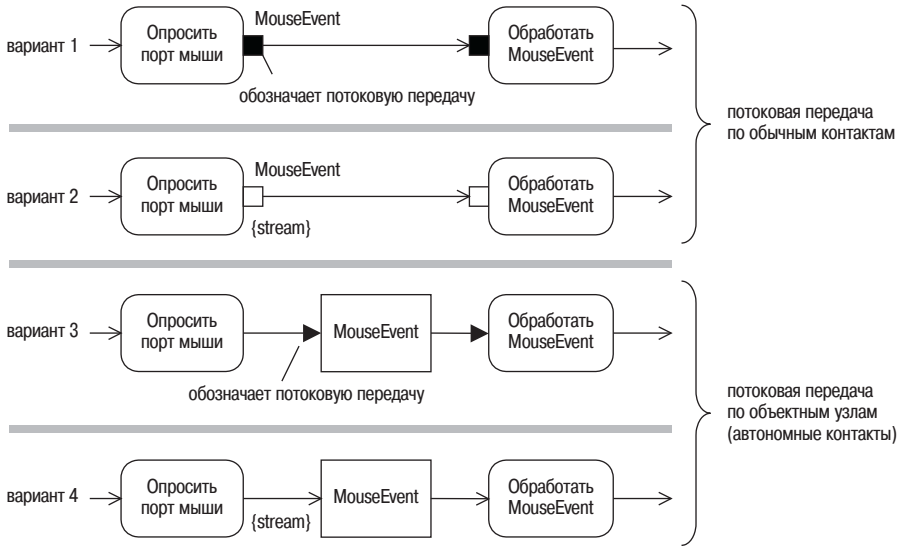


Рис. 15.11. *Четыре способа отображения потоковой передачи*

По нашему мнению, четыре способа – это слишком много для представления потоковой передачи. Рекомендуется по возможности выбрать и придерживаться только одного из них. Первый вариант является самым лаконичным – контакты закрашиваются черным; именно его мы и советуем использовать.

Пример на рис. 15.11 иллюстрирует обычное применение потоковой передачи. Действие Опросить порт мыши *непрерывно* считывает данные с порта мыши и предлагает информацию о ее деятельности в виде потока событий MouseEvent (событие мыши), передаваемого на единственное выходное ребро. Эти события принимаются действием Обработать MouseEvent.

Любая ситуация, в которой необходимо *непрерывно* получать и обрабатывать информацию, является кандидатом на использование потоковой передачи.

15.8. Дополнительные возможности потоков объектов

В этом разделе будут рассмотрены некоторые дополнительные возможности потоков объектов. Вероятно, они будут использоваться не слишком часто, но их полезно знать!

Эти возможности представлены на рис. 15.12.

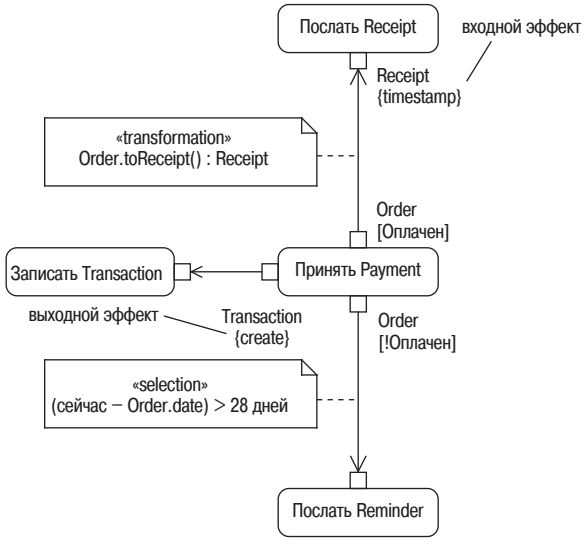


***Рис. 15.12.** Дополнительные возможности потоков объектов*

15.8.1. Входные и выходные эффекты

Входные и выходные эффекты показывают влияние, оказываемое действием на его входящие и выходящие объекты. Эти эффекты обозначаются коротким описанием в скобках, которое располагают как можно ближе к входящему или исходящему контакту.

> Входные и выходные эффекты показывают влияние, оказываемое действием на его входящие и выходящие объекты.

На рис. 15.12 рядом с действием Послать Receipt (квитанцию) можно увидеть пример входного эффекта. Он показывает, что действие Послать Receipt присваивает timestamp (временную метку) каждой получаемой им квитанции (объект Receipt). У действия Принять Payment (платеж) есть выходной эффект {create} (создать). Это означает, что Принять Payment создает все выдаваемые им объекты Transaction (транзакция).

15.8.2. Стереотип «selection»

> Выбор – это условие, накладываемое на поток объектов, согласно которому он принимает только те объекты, которые удовлетворяют этому условию.

Выбор (selection) – это условие, прикрепленное к потоку объектов, согласно которому он принимает только те объекты, которые удовлетво-

ряют условию. Условие выбора отображается в узле со стереотипом «selection».

На рис. 15.12 можно увидеть выбор, прикрепленный к потоку объектов между действиями Принять Payment и Послать Reminder (напоминание). Условие выбора: (сейчас – Order.date) > 28 дней. Выходной контакт действия Принять Payment предлагает объекты Order в состоянии !Оплачено, а выбор принимает только те объекты Order, которые ожидают оплаты более 28 дней. В результате этот поток выбирает все объекты Order, остающиеся неоплаченными дольше 28 дней, и передает их действию Послать Reminder.

15.8.3. Стереотип «transformation»

Преобразование меняет типы объектов в потоке объектов. Выражение преобразования представлено в узле, отмеченном стереотипом «transformation» (преобразование).

На рис. 15.12 преобразование можно увидеть на потоке объектов между действиями Принять Payment и Послать Receipt. Оно определяет, что объекты Order, выдаваемые действием Принять Payment, будут преобразованы в объекты Receipt, ожидаемые действием Послать Receipt. В этом примере преобразование осуществляется путем вызова операции toReceipt() для каждого объекта Order при его прохождении по ребру. Данная операция принимает информацию объекта Order и создает объект Receipt.

> Преобразование меняет типы объектов в потоке объектов.

Преобразования используются, когда необходимо соединить выходной контакт экземпляров одного классификатора с входным контактом, ожидающим экземпляры другого классификатора.

15.9. Групповая рассылка и групповой прием

> При групповой рассылке объекты рассылаются многим получателям.

Обычно объект имеет только одного получателя. Однако иногда необходимо показать, что объект посылается нескольким получателям. Это называется *групповой рассылкой* (*multicast*). Аналогично может понадобиться представить получение объектов от нескольких отправителей. Это называется *групповым приемом* (*multireceive*). Групповые рассылка и прием часто присутствуют в симметричных парах, как показано на рис. 15.13.

> При групповом приеме осуществляется прием объектов от многих отправителей.

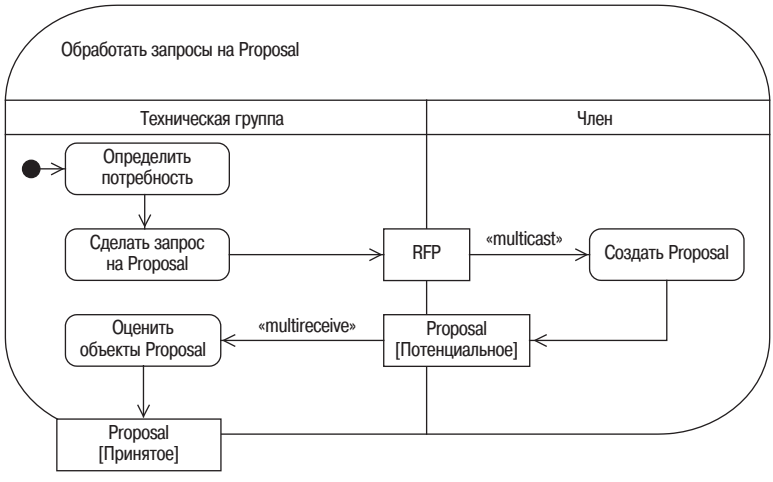


Рис. 15.13. *Групповая рассылка и групповой прием*

Чтобы показать, что поток объектов возникает из групповой рассылки или является результатом группового приема, его обозначают стереотипом «multicast» и «multireceive» соответственно. На рис. 15.13 представлен простой бизнес-процесс, аналогичный тому, который используется OMG для сбора предложений по поводу стандартов. Сначала Техническая группа устанавливает потребность в решении и формулирует ее в виде RFP (Request For Proposal – запрос на предложение). Оно рассылается многим Членам команды, которые вырабатывают потенциальные предложения (объекты Proposal). Через групповой прием Техническая группа получает эти предложения, оценивает их и выдает Принятые Proposal.

15.10. Наборы параметров

> Наборы параметров обеспечивают действию альтернативные наборы входных и выходных контактов.

Наборы параметров обеспечивают действию *альтернативные наборы* (*alternative sets*) входных и выходных контактов. Эти наборы контактов называют *наборами параметров* (*parameter sets*). Наборы входных параметров содержат входные контакты, а наборы выходных параметров содержат выходные контакты. Смешанный набор входных и выходных параметров невозможен.

Чтобы проиллюстрировать, зачем могут понадобиться наборы параметров, рассмотрим рис. 15.14, на котором представлены три типа аутентификации.

1. Получить UserName и Password – выдает объекты UserName и Password.

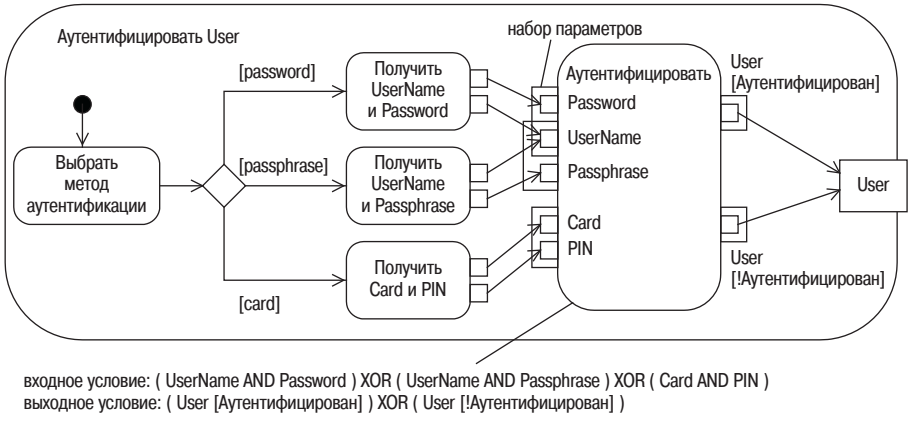


***Рис. 15.14.** Использование наборов параметров*

2. Получить UserName и Passphrase – выдает объекты UserName и Passphrase (идентификационная фраза).
3. Получить Card и PIN – выдает объекты Card и PIN.

Как видите, каждое из этих действий выдает разный набор объектов. Одним из решений этой проблемы могло бы быть разделение аутентификации на три действия для каждого набора объектов, например AuthenticateUserNameAndPassword, AuthenticateUserNameAndPassphrase и AuthenticateCardAndPIN. Это разумное решение, но оно имеет несколько недостатков.

- Оно делает диаграмму деятельности слишком большой.
- Аутентификация распределена на три действия, а не локализована в одном.

Применяя наборы параметров, можно создать *единственное* действие Аутентифицировать, способное обрабатывать разные наборы входных параметров.

На рис. 15.14 представлено действие Аутентифицировать с тремя наборами входных и двумя наборами выходных параметров. Наборы входных параметров:

- UserName AND Password;
- UserName AND Passphrase;
- Card AND PIN.

При каждом выполнении этого действия может использоваться только один из этих наборов входных параметров. Поэтому между ними установлено отношение исключающее ИЛИ (XOR).

Обратите внимание, что в наборах {UserName, Password} и {UserName, Passphrase} есть общий контакт UserName. Как показано на рис. 15.14, это можно обозначить путем наложения двух наборов параметров таким образом, чтобы общие контакты находились в области пересечения. Од-

нако в этом случае очень легко запутаться, поэтому можно использовать два отдельных контакта UserName, по одному для каждого набора входных параметров. Мы предпочитаем второй вариант решения. На приведенной диаграмме (рис. 15.14) проиллюстрирован синтаксис наложения контактов на тот случай, если вы решите их использовать.

Действие Аутентифицировать имеет следующие наборы выходных параметров, каждый из которых содержит всего один контакт:

- User[Аутентифицирован];
- User[!Аутентифицирован].

Опять же при каждом выполнении деятельности будет использоваться только один из этих наборов выходных параметров.

15.11. Узел «centralBuffer»

> Центральный буфер – это объектный узел, используемый *исключительно* в качестве буфера.

Как упоминалось в разделе 14.9.1, все объектные узлы обладают способностью буферизации.

Центральный буфер – это объектный узел, используемый *исключительно* как буфер между входными и выходными потоками объектов. Он позволяет объединять несколько входных потоков объектов и распределять объекты между несколькими выходными потоками объектов. Когда объектный узел используется как центральный буфер, он должен быть обозначен стереотипом «centralBuffer».

На рис. 15.15 показан простой пример, в котором центральный буфер используется для накопления различных типов объектов Order, непрерывно поступающих по нескольким каналам продаж. Объекты Order находятся в буфере в ожидании принимающего их действия Обработать Order. В центральном буфере Order могут сохраняться объекты типа Order или его подклассов: WebOrder (веб-заказ), PhoneOrder (заказ по телефону) и PostOrder (заказ по почте).

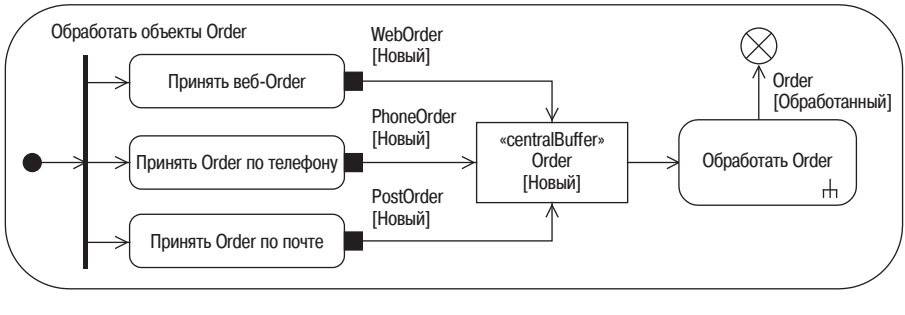


Рис. 15.15. *Центральный буфер для хранения объектов*

15.12. Диаграммы обзора взаимодействий

Диаграммы обзора взаимодействий – особая форма диаграммы деятельности. Они показывают взаимодействия и включения взаимодействий (interaction occurrences) и используются для моделирования высокоуровневого потока управления между взаимодействиями. Взаимодействия и диаграммы взаимодействий подробно обсуждаются в главе 12.

Одно особенно мощное применение диаграмм обзора взаимодействий – иллюстрация потока управления между прецедентами. Если каждый прецедент представить как взаимодействие, синтаксис диаграммы деятельности можно использовать для отображения движения потока управления между ними.

> Диаграммы обзора взаимодействий – это диаграммы деятельности, представляющие взаимодействия и включения взаимодействий.

На рис. 15.16 представлена диаграмма обзора взаимодействий ManageCourses (организация курсов), которая показывает поток между низкоуровневыми взаимодействиями LogOn (войти), GetCourseOption (получить варианты курсов), FindCourse (найти курс), RemoveCourse (удалить курс) и AddCourse (добавить курс). Каждое из этих взаимодействий представляет прецедент. Таким образом, диаграмма обзора взаимодействий фиксирует поток управления между прецедентами.

Обратите внимание, что линии жизни, принимающие участие во взаимодействии, могут быть перечислены после ключевого слова lifelines (линии жизни) в заголовке диаграммы. Такое документирование может быть полезным, поскольку часто линии жизни скрыты внутри включений взаимодействий.

Синтаксис диаграмм обзора взаимодействий аналогичен синтаксису диаграмм деятельности, за исключением того, что здесь отображаются встроенные взаимодействия и включения взаимодействий, а не деятельности и объектные узлы. Можно показать ветвление, параллельность и цикличность, как описано в табл. 15.1. Данная таблица также предоставляет обзор различий между синтаксисом диаграммы последовательностей и диаграммы обзора взаимодействий.

Таблица 15.1

Действие	Диаграммы последовательностей	Диаграмма обзора взаимодействий
Ветвление	Комбинированные фрагменты alt и opt (раздел 12.10.1)	Узлы принятия решения и слияния (раздел 14.8.2)
Параллельность	Комбинированный фрагмент par (раздел 20.5)	Узлы ветвления и объединения (раздел 14.8.3)
Итерация	Комбинированный фрагмент loop (раздел 12.11.1)	Циклы на диаграмме

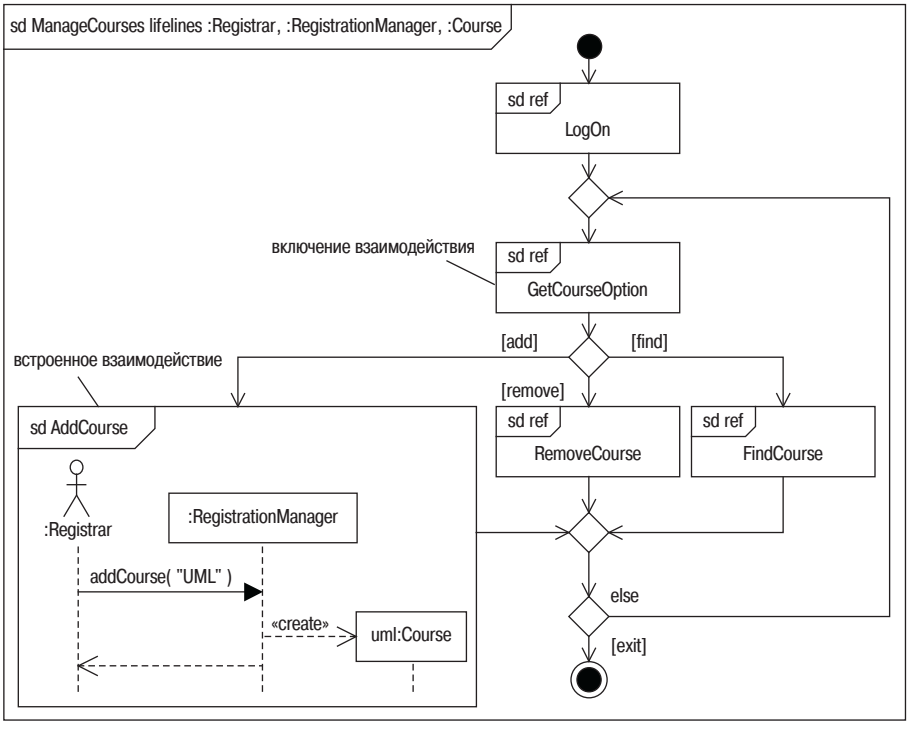


Рис. 15.16. *Диаграмма обзора взаимодействий*

На диаграммах последовательностей может быть представлено все то же самое, что и на диаграммах обзора взаимодействий. Может возникнуть вопрос, зачем нужны последние? Причина в том, что диаграммы обзора взаимодействий дают визуальное представление ветвления, параллелизма и итерации, что намного понятней и проще для восприятия. В то же время они переносят ваше внимание с элементов, например линий жизни, непосредственно на поток управления. Поэтому эти диаграммы должны использоваться, когда необходимо акцентировать внимание на движении потока управления через множество взаимодействий.

В реализации прецедента взаимодействия описывают поведение, определенное в прецеденте. Таким образом, диаграммы обзора взаимодействий могут использоваться для иллюстрации бизнес-процессов, охватывающих прецеденты.

15.13. Что мы узнали

В данной главе были представлены некоторые дополнительные возможности диаграмм деятельности. Мы изучили следующее:

- Области с прерываемым выполнением действий:
  - прерываются, когда маркер проходит по прерывающему ребру;

    - когда область прерывается, все ее потоки немедленно прекращаются;
    - прерывающие ребра отображаются в виде зигзагообразной стрелки или обычной стрелки с пиктограммой зигзага над ней.
- Контакты исключений:
    - выводят объект исключения из действия;
    - обозначаются равносторонним треугольником.
- Защищенные узлы:
    - имеют прерывающее ребро, ведущее к обработчику исключения;
    - немедленно прекращают выполнение при формировании соответствующего типа исключения и поток передается в обработчик исключения;
    - являются мгновенными (instantaneous).
- Узлы расширения:
    - представляют коллекцию объектов, входящую в или выходящую из области расширения;
    - область выполняется один раз для каждого входного элемента.
    - Ограничения:
        - тип выходной коллекции *должен* соответствовать типу входной коллекции;
        - типы объектов в коллекции, получаемой на входе, и в выходной коллекции *должны* быть одинаковыми.
    - Режимы:
        - iterative – все элементы входной коллекции обрабатываются поочередно;
        - parallel – все элементы входной коллекции обрабатываются параллельно;
        - stream – каждый элемент входной коллекции обрабатывается по мере поступления в узел;
        - нет режима, применяемого по умолчанию.
- Отправка сигналов и прием событий.
    - Сигналы:
        - информация, передаваемая между объектами асинхронно;
        - класс, обозначенный стереотипом «signal»;
        - информация хранится в атрибутах.
    - Узел действия, отправляющий сигнал:
        - инициируется, когда маркер есть на всех входных ребрах;
        - выполняется – объект сигнала создается и отправляется;
        - затем завершается и предлагает на своих выходных ребрах маркеры управления.

    - Узел действия, принимающий событие:
        - инициируется входящим ребром управления *или*, если входящих ребер нет, при запуске деятельности-владельца;
        - ожидает события заданного типа:
            - выдает маркер, описывающий событие;
            - продолжает принимать события, пока выполняется деятельность-владелец;
        - для события-сигнала выходным маркером является сигнал.
- Дополнительные обозначения потока объектов:
    - входной и выходной эффекты показывают воздействие, оказываемое действием на входные и выходные объекты:
        - эффект записывается в скобках рядом с контактом;
    - выбор – условие, накладываемое на поток объектов, согласно которому он принимает только те объекты, которые удовлетворяют данному условию:
        - условие выбора помещается в прикрепленное к потоку объектов примечание, обозначенное стереотипом «selection»;
    - преобразование – меняет тип объектов потока объектов:
        - выражение преобразования помещается в прикрепленное к потоку объектов примечание, обозначенное стереотипом «transformation».
- При групповой рассылке объект отправляется многим получателям:
    - поток объектов обозначается стереотипом «multicast».
- При групповом приеме объекты приходят от многих отправителей:
    - поток объектов обозначается стереотипом «multireceive».
- Наборы параметров позволяют действию иметь альтернативные наборы входных и выходных контактов:
    - наборы входных параметров содержат входные контакты;
    - наборы выходных параметров содержат наборы выходных контактов;
    - в каждом выполнении действия может использоваться только один набор входных и один набор выходных параметров.
- Центральный буфер – это объектные узлы, используемые исключительно в качестве буферов:
    - объектный узел обозначается стереотипом «centralBuffer».
- Диаграммы обзора взаимодействий отображают поток между взаимодействиями и включениями взаимодействий:
    - ветвление – узлы принятия решения и слияния;
    - параллелизм – узлы ветвления и объединения;
    - итерация – циклы на диаграмме.

IV

Проектирование

16

Рабочий поток проектирования

16.1. План главы

В этой главе рассматривается рабочий поток проектирования в UP. Один из самых важных рассматриваемых здесь вопросов: как аналитическая модель превращается в проектную. Второй важный вопрос: как должны разрабатываться эти модели – вместе или отдельно? Эта тема обсуждается в разделе 16.3.2. Остальные разделы главы посвящены деталям и артефактам проектирования.

16.2. Рабочий поток проектирования

> Большая часть работы по проектированию осуществляется по мере перехода от фазы Уточнение к фазе Построение.

Проектирование – основная деятельность моделирования последней части фазы Уточнение и первой половины фазы Построение. Как видно из рис. 16.2, основное внимание первых итераций направлено на определение требований и анализ. По мере завершения анализа фокус моделирования перемещается на проектирование. В значительной степени анализ и проектирование могут проводиться параллельно. Однако, как мы увидим позже, важно четко различать артефакты этих двух рабочих потоков – аналитическую модель и проектную модель.

UP рекомендует не разделять функции аналитиков и проектировщиков. За разработку артефактов (таких как прецедент) – от требований через анализ и проектирование до реализации – должна отвечать одна команда. В UP основное внимание направлено не на конкретную деятельность, а на поставляемые артефакты и контрольные точки. Иными словами, главное в UP – это «цели», а не «задачи».

Основной целью анализа было построение логической модели системы, отражающей функциональность, которую должна предоставлять

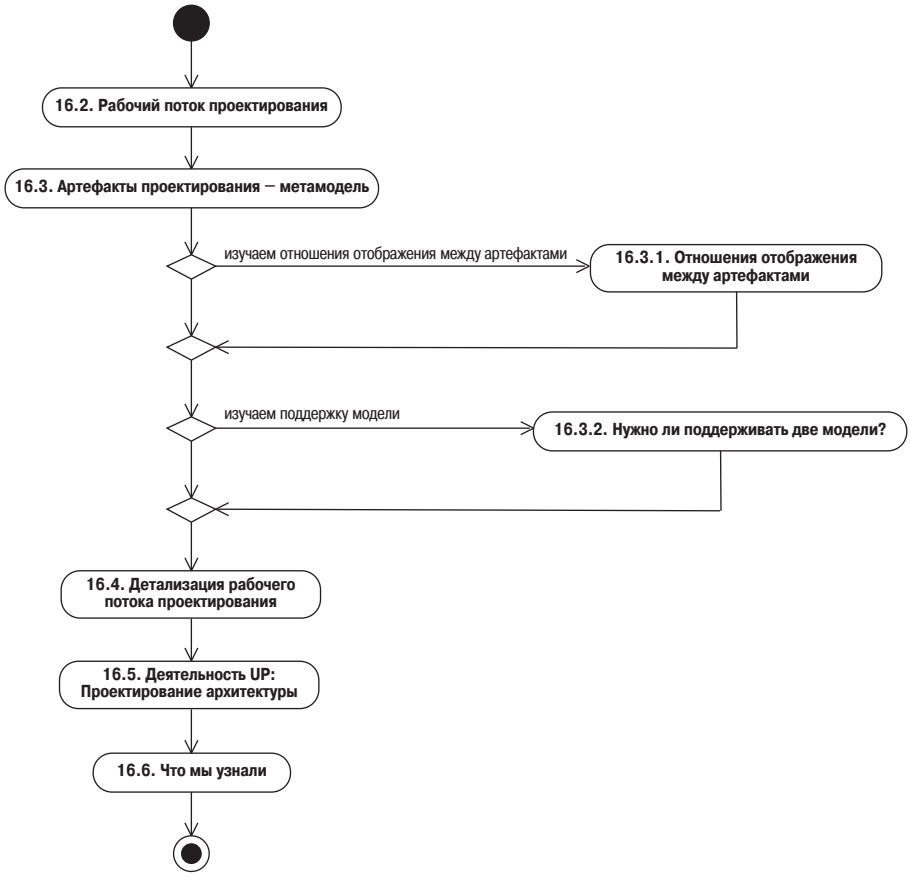


Рис. 16.1. *План главы*

система для удовлетворения требований пользователя. Цель проектирования – определить в полном объеме, как будет реализовываться эта функциональность. Одним из путей решения этой задачи является одновременное изучение предметной области и области решения. Требования поступают из предметной области, и анализ можно рассматривать как ее исследование с точки зрения заказчиков системы. Проектирование предполагает объединение технических решений (библиотек классов, механизмов сохранения объектов и т. д.) для создания модели системы (проектной модели), которая может быть реализована в действительности.

При проектировании принимаются решения по стратегическим вопросам, таким как сохранение и распределение объектов, в соответствии с которыми и создается проектная модель. Руководитель и архитектор проекта должны также разработать политики рассмотрения любых тактических вопросов проектирования.

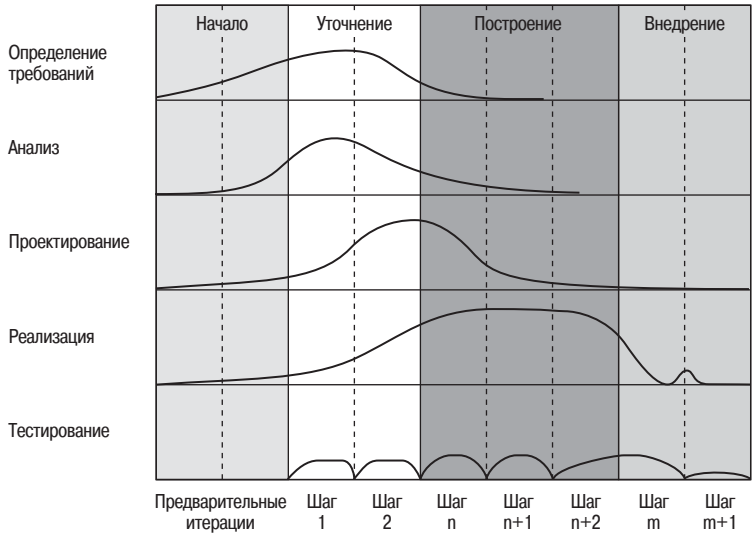


Рис. 16.2. *Процесс производства ПО (UP). Адаптировано с рис. 1.5 [Jacobson 1] с разрешения издательства Addison-Wesley*

16.3. Артефакты проектирования – метамодель

Подсистема – часть физической системы.

На рис. 16.3 представлена метамодель проектной модели. Проектная модель содержит ряд проектных подсистем (здесь показаны только две такие подсистемы). Подсистемы – это компоненты (глава 19), которые могут включать различные типы элементов модели.

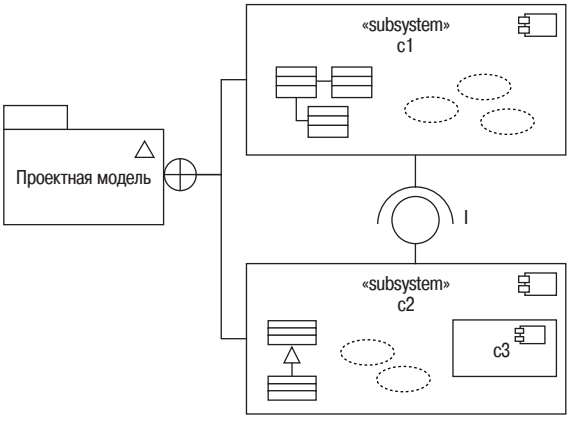


Рис. 16.3. *Метамодель проектной модели*

Хотя некоторые ключевые интерфейсы уже могли быть определены во время анализа, при проектировании им уделяется намного больше внимания. Это объясняется тем, что в конечном счете именно интерфейсы проектных подсистем объединяют систему в единое целое. Их роль при проектировании архитектуры велика, поэтому поиску и моделированию ключевых интерфейсов посвящается очень много времени. В примере на рис. 16.3 видно, что подсистеме c1 необходим интерфейс I, предоставляемый подсистемой c2. Предоставляемый и запрашиваемый интерфейсы соединяют подсистемы, как вилка и розетка. Более подробно интерфейсы обсуждаются в главе 19.

Рисунок 16.4 иллюстрирует простой пример отношения «trace» между аналитической и проектной моделями: проектная модель базируется на аналитической и может считаться просто ее улучшенной и уточненной версией.

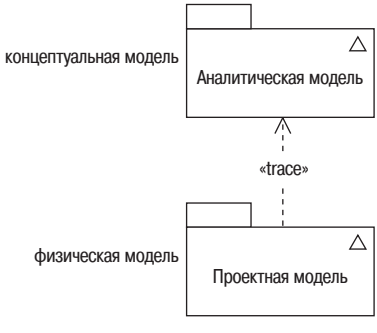


Рис. 16.4. *Проектная модель – уточненная версия аналитической модели*

Проектную модель можно рассматривать как уточнение аналитической модели с добавлением деталей и конкретных технических решений. Проектная модель содержит все то же самое, что и аналитическая, но все артефакты в ней проработаны более основательно и должны включать детали реализации. Например, аналитический класс может быть не более чем эскизом с парой атрибутов и только ключевыми операциями. Однако проектный класс должен быть совершенно точно определен – все атрибуты и операции (включая возвращаемые типы и списки параметров) должны быть полностью описаны.

Проектные модели образуются:

- проектными подсистемами;
- проектными классами;
- интерфейсами;
- реализациями прецедентов – проектными;
- диаграммой развертывания.

Одними из ключевых артефактов проектирования являются интерфейсы. В главе 19 будет показано, что они позволяют разложить систему на подсистемы, которые могут разрабатываться параллельно.

При проектировании также создается диаграмма развертывания в первом приближении, которая показывает распределение программной системы на физических вычислительных узлах. Бесспорно, эта диаграмма важна и имеет стратегическое значение. Но поскольку основная работа над диаграммой развертывания осуществляется при реализации, отложим ее обсуждение до главы 24.

16.3.1. Отношения отображения между артефактами

На рис. 16.5 показаны отношения между ключевыми артефактами анализа и проектирования.

Отношение между пакетами анализа и проектными подсистемами может быть достаточно сложным. Порой один пакет анализа будет отображаться («trace») в одну проектную подсистему, но так происходит не всегда. В силу архитектурных и технических причин один пакет анализа может быть разбит на несколько подсистем. При компонентно-ориентированной разработке проектная подсистема представляет один крупномодульный компонент. В этом случае в зависимости от требуемой детализации компонентов может оказаться, что один пакет анализа фактически раскладывается на несколько подсистем.

Класс анализа может быть превращен в один или более интерфейсов или проектных классов. Причина этого в том, что классы анализа яв-

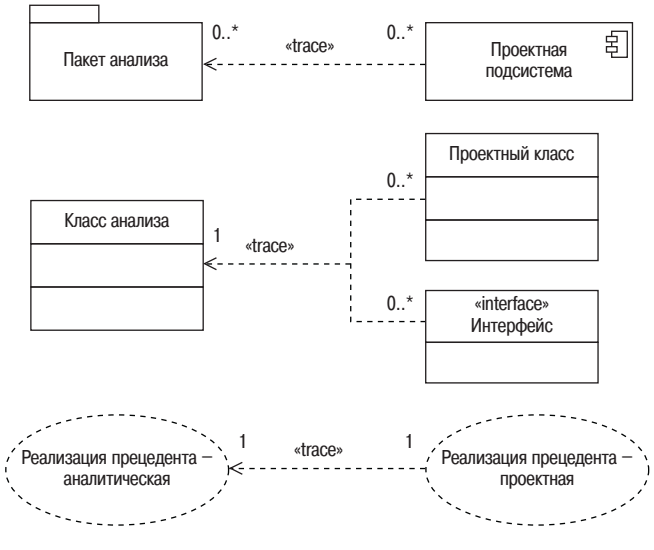


Рис. 16.5. *Отношения между ключевыми артефактами анализа и проектирования*

ляются высокоуровневым концептуальным представлением классов системы. Когда дело доходит до физического моделирования (проектирования), для реализации этих концептуальных классов может понадобиться один или более физических проектных классов и/или интерфейсов.

Между аналитическими и проектными реализациями прецедентов устанавливается простое один-к-одному отношение «trace». При проектировании реализация прецедента просто становится более детализированной.

16.3.2. Нужно ли поддерживать две модели?

В идеальном мире для системы создавалась бы единственная модель, и средство моделирования могло бы предоставлять на выбор или аналитическое, или проектное представление. Однако это требование сложнее, чем кажется на первый взгляд. Ни одно из присутствующих в настоящее время на рынке инструментальных средств UML-моделирования не может удовлетворительно справиться с задачей создания аналитического и проектного представлений на основании одной базовой модели. Похоже, нам ничего не остается, как пользоваться четырьмя стратегиями, описанными в табл. 16.1.

> Сохраняйте аналитические модели для больших, сложных или стратегически важных систем.

Таблица 16.1

	Стратегия	Результаты
1	Берется аналитическая модель и дополняется до проектной модели.	Имеется единственная проектная модель, но утеряно аналитическое представление.
2	Берется аналитическая модель, дополняется до проектной модели, а с помощью инструмента моделирования восстанавливается «аналитическое представление».	Имеется единственная проектная модель, но восстановленное инструментом моделирования аналитическое представление может быть неприемлемым.
3	Аналитическая модель замораживается в некоторой точке фазы Уточнение. До проектной модели дополняется копия аналитической модели.	Имеются две модели, но они не синхронизированы.
4	Поддерживаются две отдельные модели – аналитическая и проектная.	Имеются две модели – они синхронизированы, но их обслуживание очень трудоемко.

Идеальной стратегии нет, все зависит от проекта. Однако необходимо задать себе фундаментальный вопрос: нужно ли сохранять аналитическое представление системы? Аналитические представления обеспечи-

вают «общую картину» системы. В них может быть лишь от 1 до 10% классов подробного проектного представления, поэтому они более понятны. Их роль неоценима для следующих действий:

- введения в проект новых людей;
- понимания системы спустя месяцы или годы после ее поставки;
- понимания того, как система выполняет требования пользователей;
- обеспечения прослеживаемости требований;
- планирования обслуживания и улучшения;
- понимания логической архитектуры системы;
- привлечения внешних ресурсов к разработке системы.

Если необходимо что-то из перечисленного выше, несомненно, аналитическое представление должно быть сохранено. Обычно аналитическое представление должно сохраняться для любой большой, сложной, стратегически важной системы или системы с предположительно большим ресурсом. Это означает, что необходимо выбирать между стратегиями 3 и 4. Возможность рассинхронизации аналитической и проектной моделей должна быть всегда очень тщательно продумана. Допустимо ли это для разрабатываемого проекта?

Если система небольшая (скажем, меньше 200 проектных классов), тогда непосредственно проектная модель достаточно мала и понятна, поэтому в отдельной аналитической модели, возможно, и нет необходимости. Также, если система не имеет стратегического значения или недолговечна, разделение аналитической и проектной моделей – излишнее требование. В этом случае необходимо выбирать между стратегиями 1 и 2, и решающим фактором будут возможности используемого инструментального средства UML-моделирования. Некоторые инструменты моделирования сохраняют одну базовую модель и обеспечивают возможность применения фильтров и сокрытия информации для восстановления «аналитического» представления из проектной модели. Это разумный компромисс для многих систем среднего размера, но для очень больших систем этого, скорее всего, недостаточно.

И наконец, стоит помнить о том, что реальный срок службы многих систем существенно превышает предполагаемый!

16.4. Детализация рабочего потока проектирования

Рабочий поток UP для проектирования представлен на рис. 16.6. Основные его участники: архитектор, разработчик прецедентов и разработчик компонентов. В большинстве ОО проектов роль архитектора выполняют один или несколько специалистов, но часто они же потом являются разработчиками прецедентов и компонентов.

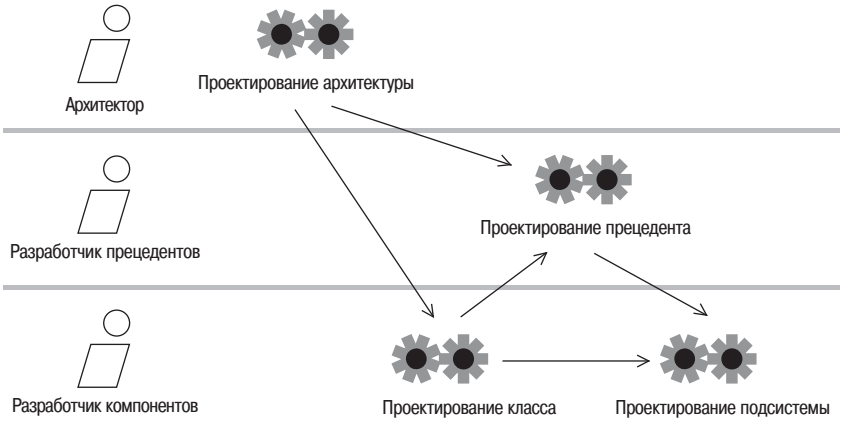


Рис. 16.6. *Рабочий поток проектирования. Воспроизведено с рис. 9.16 [Jacobson 1] с разрешения издательства Addison-Wesley*

Одна из целей UP состоит в том, чтобы определенные члены команды отвечали за отдельные части системы, начиная с анализа и заканчивая реализацией. Таким образом, специалист или команда, ответственные за создание конкретной части ОО аналитической модели, будут дополнять ее до проектной модели и, возможно с помощью программистов, превращать ее в код. При таком подходе (в этом его основное преимущество) нет «перекладывания ответственности» друг на друга между аналитиками, проектировщиками и программистами, что распространено в ОО проектах.

16.5. Деятельность UP: проектирование архитектуры

В UP начало всему процессу проектирования дает деятельность под названием Проектирование архитектуры (architectural design). Ее осуществляют один или более архитекторов. Детали этой деятельности представлены на рис. 16.7.

Как видно из рисунка, результатом Проектирования архитектуры является множество артефактов (затушеванные артефакты указывают на изменения, внесенные в оригинальный рисунок). В следующих главах этой части книги будет подробно рассмотрено, что представляют собой эти артефакты и как их найти. Самое главное, необходимо понять, что проектирование архитектуры заключается в предварительном описании артефактов, важных с архитектурной точки зрения, с целью создания общего плана архитектуры системы. Обозначенные в общих чертах артефакты являются входными данными для более детального проектирования, в процессе которого происходит их конкретизация.

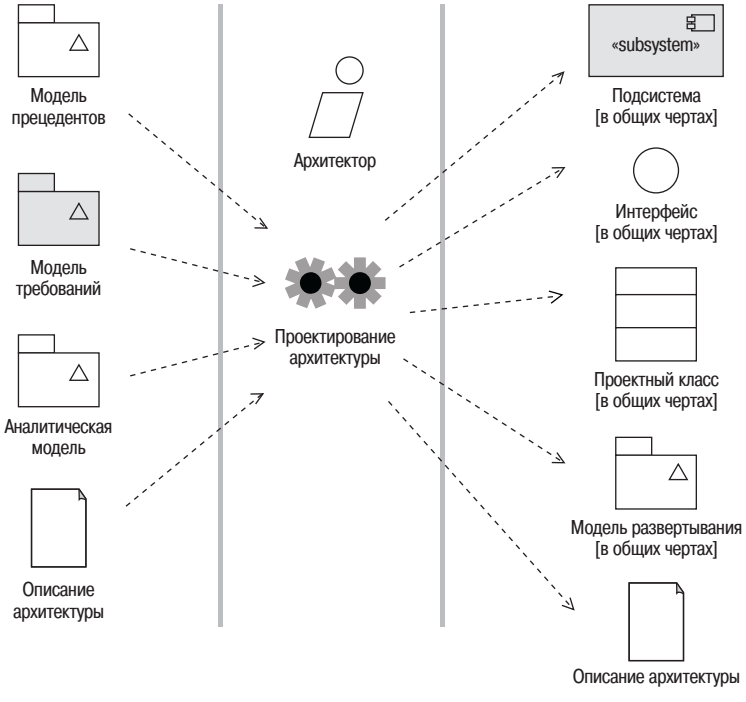


Рис. 16.7. *Деятельность UP Проектирование архитектуры. Адаптировано с рис. 9.17 [Jacobson 1] с разрешения издательства Addison-Wesley*

Обычно Проектирование архитектуры не выделяется в отдельный шаг. Не следует забывать, что UP – итеративный процесс. Таким образом, проработка деталей архитектуры системы ведется на *всем протяжении* завершающих этапов Уточнения и в начале Построения.

16.6. Что мы узнали

В рабочем потоке проектирования определяется, как будет реализовываться функциональность, описанная в аналитической модели. Мы узнали следующее:

- Проектирование – основная деятельность при моделировании в последней части фазы Уточнение и первой части фазы Построение.
  - Анализ и проектирование в некоторой степени могут происходить параллельно.
  - Одна команда должна провести артефакт от анализа до проектирования.
  - ОО проектировщики основное внимание должны уделять главнейшим вопросам проектирования, таким как арихитектуры распределенных компонентов – политики и стандарты должны

вводиться для решения тактически важных вопросов проектирования.

- Проектная модель включает:
  - проектные подсистемы;
  - проектные классы;
  - интерфейсы;
  - реализации прецедентов – проектные;
  - диаграмму развертывания (в первом приближении).
- Отношения отображения существуют между:
  - проектной и аналитической моделями;
  - одной или более проектными подсистемами и пакетом анализа.
- Следует поддерживать две отдельные модели, аналитическую и проектную, если система:
  - большая;
  - сложная;
  - стратегически важная;
  - подвержена частым изменениям;
  - предположительно с большим сроком службы;
  - для ее разработки привлекаются внешние ресурсы.
- Деятельность UP Проектирование архитектуры – это итеративный процесс, имеющий место в конце фазы Уточнение и в начале фазы Построение:
  - в ходе процесса создаются и описываются в общих чертах артефакты, которые в дальнейшем конкретизируются.

17

Проектные классы

17.1. План главы

В этой главе рассказывается о проектных классах – строительных блоках проектной модели. Для ОО проектировщика жизненно важно понимать, как моделировать эти классы эффективно.

После описания контекста UP мы рассмотрим анатомию проектного класса, а затем в разделе 17.5 перейдем к анализу того, что образует правильно сформированный проектный класс. Здесь будут рассмотрены требования полноты и достаточности, простоты, высокой внутренней связности, низкой связанности с другими классами и применимости агрегации вместо наследования.

17.2. Деятельность UP: Проектирование класса

В детализации рабочего потока проектирования в UP (рис. 16.6) после Проектирования архитектуры (Architectural design) следуют деятельности Проектирование класса (Design a class) и Проектирование прецедента (Design a use case) (раздел 20.2). Они являются параллельными и итеративными.

В этом разделе рассматривается деятельность UP Проектирование класса, представленная на рис. 17.2. Мы расширили ее и показали Интерфейс [полный] как явный результат Проектирования класса. Этот артефакт затушеван, чтобы показать, что он был изменен. В оригинальном описании деятельности он был неявным результатом.

Создание входного артефакта Класс анализа [полный] уже обсуждалось в части книги, посвященной анализу, поэтому мы больше не будем возвращаться к этому вопросу.

Стоит более подробно рассмотреть артефакт Проектный класс [в общих чертах]. С точки зрения деятельности кажется, что здесь присутствуют два отдельных самостоятельных артефакта, Проектный класс [в общих

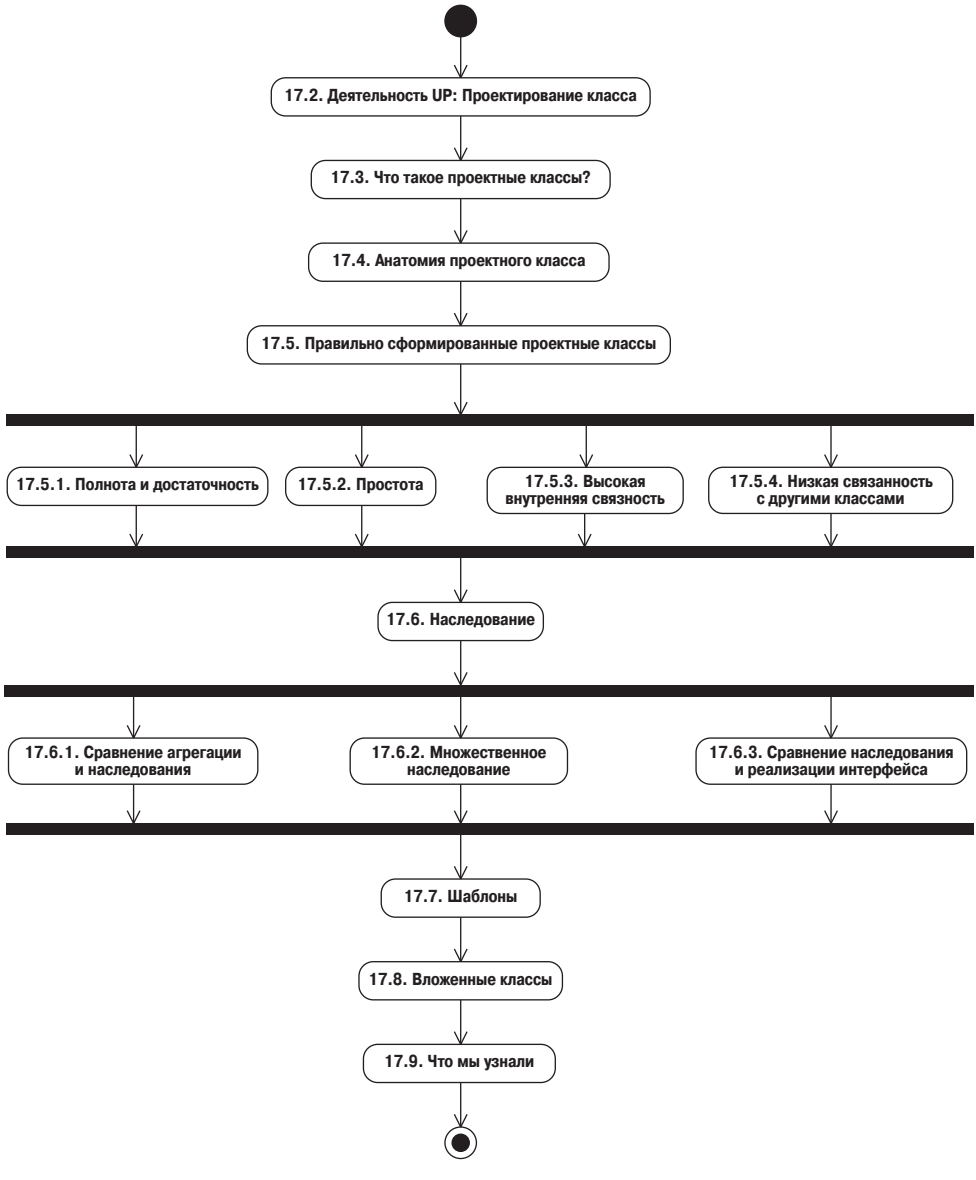


Рис. 17.1. *План главы*

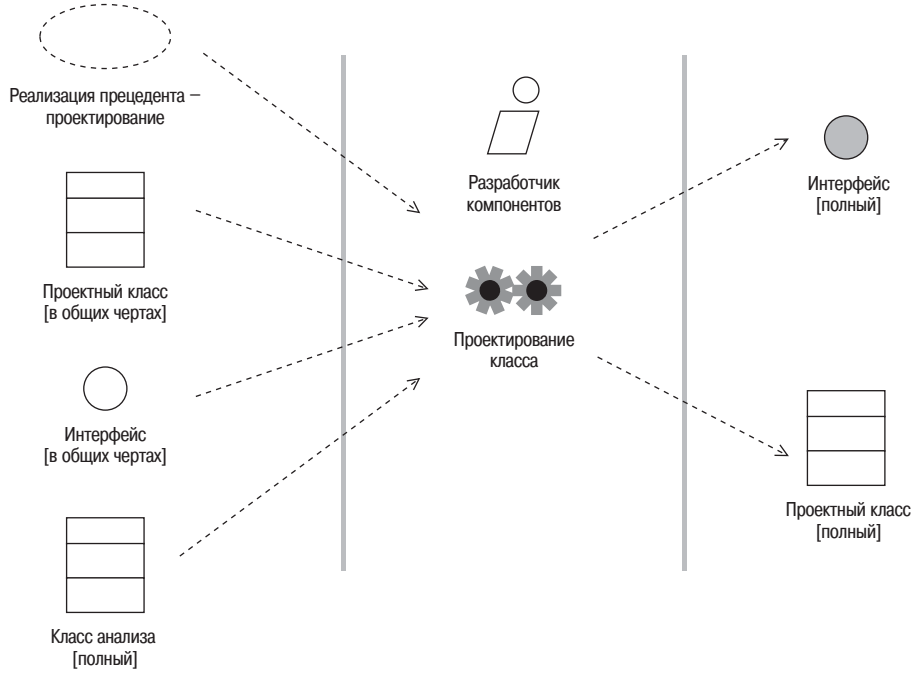


Рис. 17.2. *Деятельность Проектирование класса. Адаптировано с рис. 9.39 [Jacobson 1] с разрешения издательства Addison-Wesley*

чертах] и Проектный класс [полный]. Однако это *не* так. Они просто представляют один и тот же артефакт (проектный класс) на разных этапах развития.

Если взять набор артефактов UP-проекта в конце фазы Уточнение или в начале Построения, то не найдется артефактов под названием Проектный класс [в общих чертах] или Проектный класс [полный]. Там будут только проектные классы, находящиеся на разных этапах своего развития.

С точки зрения UP «полный» проектный класс – это класс, достаточно детализированный, чтобы служить базой для создания исходного кода. Это основное положение, о котором начинающие разработчики моделей часто забывают. Проектные классы моделируются с той степенью детализации, которая позволяет создать код, полученный на их основе. Поэтому модели проектных классов редко бывают исчерпывающими. Необходимый уровень детализации определяется проектом. Если предполагается генерировать код прямо из модели, то проектные классы необходимо моделировать очень подробно. С другой стороны, если программисты не будут использовать их в качестве рабочего прототипа, модели проектных классов могут быть менее детальными. В этой главе рассказывается, как моделировать проектные классы, достаточно подробные для любого проекта.

Соображения по поводу артефактов Проектный класс [в общих чертах] и Проектный класс [полный] также применимы к Интерфейсу [в общих чертах] и Интерфейсу [полный].

Входной артефакт Реализация прецедента – проектирование (use case realization – design) – это всего-навсего завершающий этап жизненного цикла реализации прецедента. Хотя он изображен вливающимся в Проектирование класса, на самом деле он включает проектные классы как часть своей структуры и разрабатывается параллельно с ними. Отложим обсуждение Реализации прецедента – проектирование до главы 20, поскольку, по нашему мнению, эффективнее (и понятней для читателя) сначала рассмотреть его составляющие части.

17.3. Что такое проектные классы?

Проектные классы – это классы, описания которых настолько полные, что они могут быть реализованы.

При анализе источником классов является предметная область. Это набор требований, описывающий задачу, которую необходимо решить. Как мы видели, прецеденты, описания требований, глоссарии и любая другая относящаяся к делу информация могут использоваться как источник классов анализа.

> Предметная область и область решения являются источником проектных классов.

У проектных классов два источника:

- Предметная область посредством уточнения классов анализа; это уточнение включает добавление деталей реализации. В ходе этого процесса часто обнаруживается, что высокоабстрактный класс анализа необходимо разбить на два или более детализированных проектных класса. Реализацию класса анализа описывает отношение «trace», устанавливаемое между ним и одним или более проектными классами.
- Область решения – это царство библиотек утилитных классов и многократно используемых компонентов, таких как Time, Date, String, коллекции и т. д. Здесь находится промежуточное программное обеспечение (middleware), такое как коммуникационное ПО, базы данных (и реляционные, и объектные) и компонентные инфраструктуры, например .NET, CORBA или Enterprise JavaBeans, а также средства для построения GUI. Эта область предоставляет технические инструментальные средства для реализации системы.

Это проиллюстрировано на рис. 17.3.

При анализе моделируется, *что* должна делать система. При проектировании моделируется то, *как* это поведение может быть реализовано.

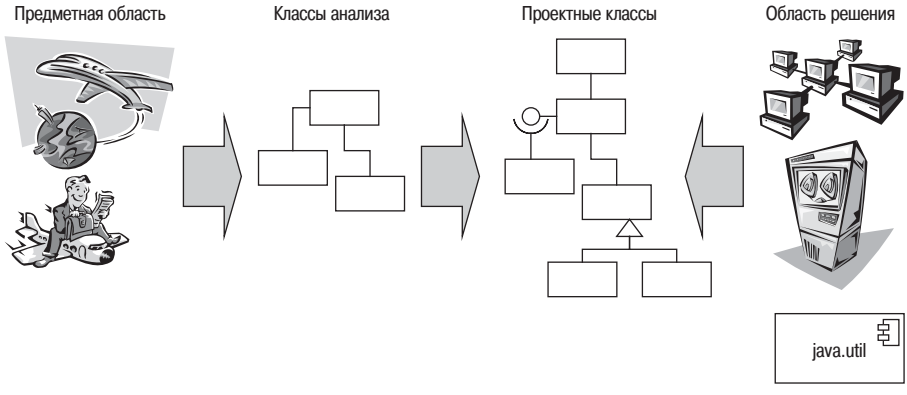


Рис. 17.3. *Два источника проектных классов: предметная область и область решения*

Почему класс анализа может быть уточнен в один или более проектных классов или интерфейсов? Это понятно, поскольку класс анализа описан на очень высоком уровне абстракции. Здесь нет полного набора атрибутов, а набор операций – это фактически только эскиз, отражающий ключевые сервисы, предлагаемые классом.

При переходе к проектированию все операции и атрибуты класса должны быть полностью описаны, поэтому нередко он становится слишком большим. Если это происходит, необходимо разбить его на два или более меньших классов. Помните, что всегда надо стремиться проектировать небольшие классы, являющиеся самодостаточными, связными элементами, которые хорошо справляются с одной-двумя функциями. Любой ценой необходимо избегать больших классов, напоминающих «швейцарский армейский нож», которые делают все.

Выбранный метод реализации определяет требуемую степень полноты описаний проектных классов. Если планируется передать модель проектного класса программистам в качестве руководства для написания кода, проектные классы должны быть полными лишь настолько, чтобы обеспечить им возможность эффективного выполнения задания. Все зависит от квалификации программистов и того, насколько хорошо они понимают предметную область и область решения. Это выясняется для каждого конкретного проекта индивидуально.

Однако если предполагается использовать проектные классы для генерации кода с помощью соответствующим образом оснащенного инструментального средства моделирования, их описания должны быть полными во всех отношениях. Генератор кода, в отличие от программиста, не может заполнять пробелы. Далее в этой главе обсуждение материала ведется исходя из предположения, что требуется очень высокая степень детализации.

17.4. Анатомия проектного класса

С помощью классов анализа делается попытка зафиксировать требуемое поведение системы без рассмотрения его возможной реализации. В проектных классах необходимо точно определить, как каждый класс будет осуществлять свои обязанности. Для этого нужно сделать следующее:

- закончить набор атрибутов и полностью описать их, включая имя, тип, видимость и (необязательно) применяемое по умолчанию значение;
- закончить набор операций и полностью описать их, включая имя, список параметров и возвращаемый тип.

Этот процесс уточнения проиллюстрирован на рис. 17.4.

Как было показано в главе 8, операция в классе анализа – это высокоуровневое логическое описание части функциональности, предлагаемой классом. В соответствующих проектных классах каждая операция класса анализа уточняется и превращается в одну или более детализированных и полностью описанных операций, которые могут быть реализованы как исходный код. Следовательно, одна высокоуровневая операция этапа анализа на самом деле может распадаться на одну или более проектных операций, которые можно реализовать. Эти детализированные операции уровня проектирования иногда называют методами.

Для иллюстрации рассмотрим следующий пример. При анализе системы регистрации авиапассажиров можно определить высокоуровневую операцию checkIn() (зарегистрировать). Однако если вам когда-либо приходилось стоять в очереди на регистрацию на рейс, то вы знаете, что это довольно сложный бизнес-процесс, включающий сбор и проверку достоверности определенной информации о пассажире, прием

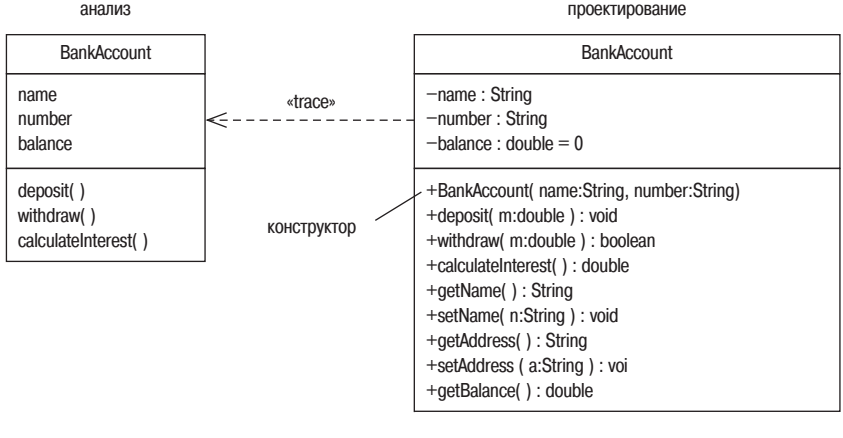


Рис. 17.4. *Детализация на этапе проектирования*

багажа и назначение посадочного места в самолете. Поэтому справедливо предполагать, что при подробном проектировании процесса высокоуровневая аналитическая операция checkIn() будет разбита на ряд операций, находящихся на более низком уровне абстракции. Высокоуровневая операция checkIn() может сохраниться, но на этапе проектирования она будет вызывать ряд «вспомогательных» операций, распределяя среди них свои обязанности. Может даже случиться так, что для реализации довольно сложного процесса регистрации необходимо будет ввести несколько новых вспомогательных классов, которые не были обозначены при анализе.

17.5. Правильно сформированные проектные классы

Проектная модель будет передана программистам для фактического создания исходного кода. Код также может генерироваться непосредственно из самой модели, если это поддерживается инструментом моделирования. Следовательно, проектные классы должны быть достаточно подробно описаны. В процессе описания определяется, является ли класс «правильно сформированным» или нет.

> Проектный класс всегда должен оцениваться с точки зрения его пользователей.

При создании важно всегда рассматривать проектный класс с точки зрения его потенциальных клиентов. Каким они увидят этот класс – не слишком ли он сложен? Может быть что-то упущено? Как тесно он взаимосвязан с другими классами? Соответствует ли его имя выполняемым функциям? Все это важно и может быть сведено в следующие четыре основные характеристики правильно сформированного проектного класса:

- полный и достаточный;
- простой;
- обладает высокой внутренней связностью;
- обладает низкой связанностью с другими классами.

17.5.1. Полнота и достаточность

> Открытые операции класса определяют контракт между классом и его клиентами.

Открытые операции класса определяют контракт между классом и его клиентами. Как и для делового контракта, важно, чтобы этот контракт был четким, правильно определенным и приемлемым для всех партнеров.

Полнота характеризует соответствие предоставляемых классом сервисов тому, что ожидают клиенты. Предположения о наборе доступных операций клиенты будут делать исходя из имени класса. Обратимся к примеру из реальной жизни: при покупке новой машины вполне разумно ожидать, что у нее будут колеса! То же самое с классами: делать вывод о том, какие операции должны быть доступны, клиенты будут из имени класса и описания его семантики. Например, от класса BankAccount (банковский счет), предоставляющего операцию withdraw(...), будут ожидать и наличия операции deposit(...). Опять-таки, если проектируется такой класс, как ProductCatalog (каталог продуктов), любой клиент может небезосновательно ожидать, что сможет добавлять, удалять и вести поиск продуктов (Product) по каталогу. Такая семантика ясно следует из имени класса. Полнота гарантирует, что классы удовлетворяют всем справедливым ожиданиям клиента.

> Полный и достаточный класс предлагает пользователям такой контракт, какой они ожидают – не больше и не меньше.

Достаточность, с другой стороны, гарантирует, что все операции класса полностью сосредоточены на реализации его предназначения. Класс никогда не должен удивлять клиента. Он должен содержать только ожидаемый набор операций, не более этого. Например, обычная ошибка новичков: взять простой достаточный класс, такой как BankAccount, и затем добавить в него операцию по обработке кредитных карт или по управлению политиками страхования и т. д. Достаточность заключается в максимально возможном сохранении простоты и узкой специализированности проектного класса.

Золотое правило обеспечения полноты и достаточности: класс должен делать то, что ожидают от него пользователи, не больше и не меньше.

17.5.2. Простота

Операции должны проектироваться таким образом, чтобы предлагать единственный, простой, элементарный сервис. Класс *не* должен предлагать множество способов выполнения одного и того же, поскольку это запутывает клиентов и может привести к усложнению технического обслуживания и проблемам совместимости.

> Простота – сервисы должны быть простыми, элементарными и уникальными.

Например, если в классе BankAccount есть простая операция создания одного депозитного вклада, в нем *не* должно быть более сложных операций, создающих два или более депозитов. Такого же эффекта можно добиться, повторяя простую операцию. Класс всегда должен предлагать самый простой и самый малый набор возможных операций.

Хотя придерживаться простоты – хорошее правило, тем не менее возможны обстоятельства, вынуждающие от него отойти. Самой распространенной причиной нарушения ограничения простоты является повышение производительности. Например, если групповое создание депозитных вкладов по сравнению с одиночным существенно повышает производительность, можно ослабить требование к простоте и ввести в класс BankAccount более сложную операцию deposit(...), обрабатывающую сразу несколько транзакций. Однако отправной точкой в проектировании *всегда* должен быть самый простой из возможных набор операций. Усложнение должно происходить, только если для этого есть веские и обоснованные причины.

Это важный момент. Многие из так называемых проектных «оптимизаций» в большей степени основываются на вере, а не на твердых фактах. В результате их влияние на фактическую производительность времени выполнения приложения мало или вообще отсутствует. Например, если приложение будет проводить в данной операции всего 1% своего времени, оптимизация этой операции может ускорить приложение не более чем на 1%. Полезное практическое правило: большинство приложений проводит около 90% времени выполнения в 10% своих операций. Вот эти операции и надо находить и оптимизировать, чтобы получить реальное повышение производительности. Такую настройку производительности можно осуществить только с помощью инструмента профилирования кода, такого как jMechanic для Java (*http://jmechanic.sourceforge.net*), который проводит сбор параметров производительности исполняющегося кода. Конечно, это задача реализации, которая может влиять на проектную модель.

17.5.3. Высокая внутренняя связность

Каждый класс должен моделировать только одно абстрактное понятие и иметь набор операций, обеспечивающих предназначение класса. Это и есть связность (cohesion). Если необходимо, чтобы у класса было множество разных обязанностей, для реализации некоторых из них можно создать «вспомогательные классы». Тогда основной класс может делегировать обязанности своим «помощникам».

Внутренняя связность – одна из самых желательных характеристик класса. Связные классы обычно проще понимать, повторно использовать и обслуживать. У связного класса небольшой набор тесно взаимосвязанных обязанностей. Каждая операция, атрибут и ассоциация класса специально проектируются для реализации этого маленького, узкоспециализированного набора обязанностей.

> Каждый класс должен отражать единственную четко определенную абстракцию, используя минимальный набор возможностей.

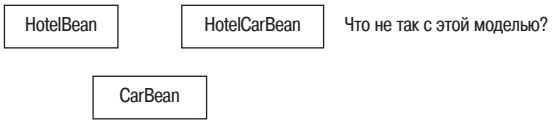


***Рис. 17.5.** Неверная модель системы*

Как-то нам попалась на глаза модель системы продаж, приведенная на рис. 17.5, которая несколько смутила нас. В ней есть класс HotelBean (компонент гостиница), класс CarBean (компонент машина) и класс HotelCarBean (компонент машина-гостиница) (компонент (bean) – это корпоративный компонент Java (Enterprise JavaBeans, EJBs)). HotelBean отвечал за сдачу комнат в гостиницах, CarBean – за прокат автомобилей, а HotelCarBean – за продажу пакета этих услуг (прокат автомобиля при остановке в гостинице). Очевидно, что эта модель неверна по нескольким причинам.

- Имена классов подобраны неверно – HotelStay (остановка в гостинице) и CarHire (прокат автомобиля) подошли бы намного лучше.
- Суффикс «Bean» не нужен, поскольку он просто указывает на определенную деталь реализации.
- Класс HotelCarBean имеет очень слабую связность – его две основные обязанности (сдача гостиничных номеров и прокат автомобилей) уже выполняются двумя другими классами.
- Это и не аналитическая модель (в ней есть информация проектирования – суффиксы «Bean»), и не проектная модель (она недостаточно полная).

С точки зрения внутренней связности классы HotelBean и CarBean более или менее приемлемы (при условии, что будут переименованы), но HotelCarBean просто абсурден.

17.5.4. Низкая связанность с другими классами

Конкретный класс должен быть ассоциирован *ровно* с таким количеством классов, *которого достаточно* для того, чтобы он мог реализовывать свои обязанности. Взаимоотношения должны устанавливаться только в том случае, если между классами существует реальная семантическая связь – это обеспечивает низкую связанность (coupling).

> Класс должен быть ассоциирован с минимальным количеством классов, позволяющим ему реализовывать свои обязанности.

Одна из распространенных ошибок неопытных ОО проектировщиков – объединение в модели всего со всем практически случайным образом. По сути, связанность – самый злостный ваш враг в объектном моделировании, поэтому необходимо заранее принимать меры, чтобы ограни-

чить отношения между классами и максимально сократить связанность.

Объектная модель с большим количеством взаимосвязей эквивалентна «спагетти-коду» в не-ОО мире. Она приведет к созданию малопонятной и неудобной в эксплуатации системы. Вы увидите, что ОО системы с большим количеством взаимосвязей часто возникают в проектах, в которых нет формального процесса моделирования, а система просто эволюционирует во времени произвольным образом.

Неопытные проектировщики должны быть внимательными и *не* устанавливать связи между классами просто потому, что в одном из них есть код, который может использоваться другим. Это самый плохой вариант повторного использования, когда ради небольшой экономии времени разработки в жертву приносится архитектурная целостность системы. Все ассоциации между классами должны тщательно продумываться. Многие ассоциации перейдут в проектную модель непосредственно из аналитической, но целый ряд ассоциаций вводится из-за ограничений реализации или желания повторно использовать код. Эти ассоциации необходимо проверять наиболее внимательно.

Конечно, некоторая степень связанности хороша и необходима. Допускается высокая связанность в рамках подсистемы, поскольку это указывает на высокую связность компонента. Подрывают архитектуру только многочисленные связи *между* подсистемами. Необходимо активно стремиться к сокращению подобной связанности.

17.6. Наследование

При проектировании наследование играет намного более важную роль, чем при анализе. В анализе наследование использовалось, *только* если между классами анализа имело место четкое и явно выраженное отношение «является». Однако при проектировании наследование может применяться в тактических целях для повторного использования кода. Это совсем иная стратегия, поскольку наследование фактически используется для упрощения реализации дочернего класса, а не для выражения бизнес-отношения между родителем и потомком.

В следующих нескольких разделах рассматриваются стратегии эффективного использования наследования в проектировании.

17.6.1. Сравнение агрегации и наследования

Наследование – очень мощный метод. Оно является ключевым механизмом формирования полиморфизма в строго типизированных языках программирования, таких как Java, C# и C++. Однако неопытные ОО проектировщики и программисты часто используют его неправильно. Необходимо осознавать, что наследование имеет определенные нежелательные характеристики.

> Наследование – самая строгая форма связанности классов. Это жесткое отношение.

- Это самая строгая из возможных форма связанности двух или более классов.
- В иерархии классов инкапсуляция низкая. Изменения базового класса передаются вниз по иерархии и приводят к изменениям подклассов. Это явление называют проблемой «хрупкости базового класса», когда изменения базового класса имеют огромное влияние на другие классы системы.
- Это очень жесткий тип отношения. Во всех широко используемых ОО языках программирования отношения наследования постоянны во время выполнения. Создавая и уничтожая отношения во время выполнения, можно изменять иерархии агрегации и композиции, но иерархии наследования остаются неизменными. Это делает наследование самым жестким типом отношений между классами.

Система на рис. 17.6 – типичный пример решения задачи моделирования ролей в организации, выполненного неопытным разработчиком. На первый взгляд все вполне приемлемо, однако здесь есть проблемы. Рассмотрим предложенный вариант. Объект john типа Programmer (программист) необходимо повысить до типа Manager (менеджер). Вам должно быть понятно, что изменить класс Джона (john) во время выполнения нельзя. Поэтому единственный способ повысить Джона – создать новый объект Manager (названный john:Manager), скопировать в него из объекта john:Programmer все существующие данные и затем удалить john:Programmer для сохранения целостности приложения. Конечно, это очень сложно и совершенно не соответствует тому, как все происходит на самом деле.

В сущности, в модели на рис. 17.6 допущена фундаментальная семантическая ошибка. Служащий (Employee) – это *именно* должность (Job) или это указывает на то, что служащий *занимает* некоторую должность? Ответ на этот вопрос приводит к решению проблемы (см. рис. 17.7).

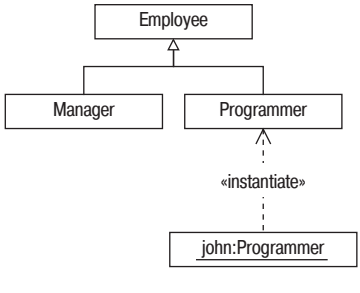


***Рис. 17.6.** Неверная модель*

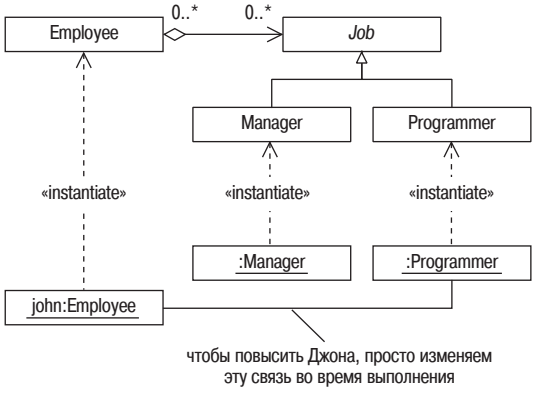


Рис. 17.7. *Верная модель*

При использовании агрегации получаем верную семантику – у служащего (Employee) *есть* должность (*Job*). При такой более гибкой модели у служащих при необходимости может быть несколько должностей.

> Подклассы всегда должны представлять «особую разновидность чего-либо», а не «выполняемую роль».

Путем замены наследования агрегацией получена более гибкая и семантически правильная модель. Здесь представлен важный общий принцип: подклассы всегда должны «являться разновидностью чего-либо», а не «являться ролью, исполняемой чем-либо». Если рассматривается бизнес-семантика компаний, служащих и должностей, очевидно, что должность – это *роль*, *исполняемая* служащим, и она на самом деле не указывает на *разновидность* служащего. Таким образом, наследование – безусловно, неверный выбор для моделирования такого рода бизнес-отношения. С другой стороны, в компании много *разных* должностей. Это указывает на то, что иерархия наследования должностей (корнем которой является абстрактный базовый класс *Job*), вероятно, является хорошей моделью.

17.6.2. Множественное наследование

> При множественном наследовании у класса может быть более одного родителя.

Иногда возникает потребность наследования от нескольких родителей. Такое наследование называют множественным. Его поддерживают не все ОО языки программирования, например в Java и C# допускается только единичное наследование. На практике отсутствие поддержки множественного наследования не является проблемой, по-

скольку его всегда можно заменить единичным наследованием и делегированием. Даже несмотря на то, что иногда множественное наследование предлагает более элегантное решение задачиы проектирования, оно может использоваться, *только* если целевой язык реализации его поддерживает.

О множественном наследовании важно знать следующее.

> Родители должны быть семантически не связанными.

- Все участвующие родительские классы должны быть семантически не связанными. В случае наличия какого-либо совмещения в семантике базовых классов между ними возможны непредвиденные взаимодействия. Это может привести к необычному поведению подкласса. Мы говорим, что базовые классы должны быть ортогональными (расположенными под прямым углом друг относительно друга).
- Между подклассом и всеми его суперклассами должны действовать принцип замещаемости и принцип «является разновидностью».
- У суперклассов не должно быть общих родителей. В противном случае в иерархии наследования образуется цикл и возникает множество путей наследования одних и тех же возможностей от более абстрактных классов. У языков программирования, поддерживающих множественное наследование (таких как C++), есть специальные, особые для каждого языка способы разрешения циклов в иерархии наследования.

> Смешанные классы создаются, чтобы путем множественного наследования быть «смешанными» с другими классами. Это надежное и мощное средство.

Один из общепринятых способов эффективного использования множественного наследования – «смешанный» (mixin) класс. Эти классы не являются по-настоящему автономными классами. Они разрабатываются специально для того, чтобы быть «смешанными» с другими классами с помощью наследования. На рис. 17.8 класс Dialer (набиратель номера) – простой смешанный класс. Его единственная функция – набор телефонного номера. То есть сам по себе он не представляет особой ценности, но предоставляет связный пакет полезного поведения,

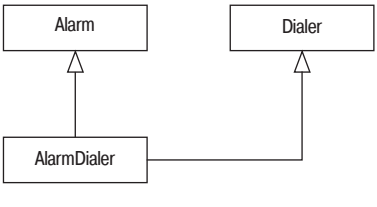


***Рис. 17.8.** Смешанный класс Dialer*

которое может широко использоваться другими классами посредством множественного наследования. Этот смешанный класс – пример обычного утилитного класса, который мог бы стать частью многократно используемой библиотеки.

17.6.3. Сравнение наследования и реализации интерфейса

При наследовании мы получаем две вещи:

- интерфейс – открытые операции базовых классов;
- реализацию – атрибуты, отношения, защищенные и закрытые операции базовых классов.

При реализации интерфейса (глава 19) получаем только одно:

- интерфейс – набор открытых операций, атрибутов и отношений, *не* имеющих реализации.

У наследования и реализации интерфейса есть что-то общее, поскольку оба этих механизма обеспечивают возможность описания контракта, который должен быть реализован подклассами. Однако семантика и применение этих двух методов совершенно разные.

Наследование должно использоваться только тогда, когда речь идет о наследовании от суперкласса некоторых деталей реализации (операций, атрибутов, отношений). Это одна из разновидностей многократного использования. На заре ОО наследование часто считали основным механизмом многократного использования. Однако мир не стоит на месте, и разработчики выявили подчас неприемлемые ограничения, накладываемые наследованием. Поэтому от его применения несколько отошли.

Реализация интерфейса полезна везде, где необходимо определить контракт, но *без* наследования деталей реализации. Хотя реализация интерфейса фактически не обеспечивает повторного использования кода, она предоставляет безупречный механизм описания контрактов и гарантирует их выполнение классами реализации. Поскольку в реализации интерфейса ничего не наследуется, это в некотором отношении более гибкий и надежный механизм, чем наследование.

17.7. Шаблоны

До сих пор при описании проектного класса нам приходилось явно задавать типы атрибутов, возвращаемые типы всех операций и типы всех параметров операций. Это нормально и хорошо работает в большинстве случаев, но иногда может ограничивать возможность повторного использования кода.

В примере на рис. 17.9 описаны три класса. Все три являются ограниченными массивами. Один из массивов типа int, следующий – double и последний – String. Внимательно посмотрев на эти классы, можно за-

BoundedIntArray	BoundedDoubleArray	BoundedStringArray
size : int elements[ ] : int	size : int elements[ ] : double	size : int elements[ ] : String
addElement(e:int) : void getElement(i:int) : int	addElement(e:double) : void getElement(i:int) : double	addElement(e:String) : void getElement(i:int) : String

Рис. 17.9. *Три класса отличаются только типом элементов*

метить, что они идентичны, за исключением типа элементов, хранящихся в массиве. Однако, несмотря на это сходство, были определены три разных класса.

Шаблоны (templates) позволяют параметризовать тип. Это означает, что вместо определения фактических типов атрибутов, возвращаемых значений операций и параметров операций можно описать класс с помощью структурных нулей (placeholder), или параметров. При создании новых классов они могут быть заменены фактическими значениями.

На рис. 17.10 класс BoundedArray описан с помощью параметров type (тип) (который по умолчанию является классификатором) и size (размер) типа int. Обратите внимание, что шаблон определяет для size применяемое по умолчанию значение 10. Это значение используется, если при создании экземпляра шаблона размер *не* будет задан.

Связывая конкретные значения с этими формальными параметрами, можно создавать новые классы. Это называют созданием экземпляра шаблона. Заметьте, что в результате создания экземпляра шаблона получается класс, экземпляры которого можно затем создавать для получения объектов.

Как показывает рис. 17.10, экземпляр шаблона можно создать с помощью отношения реализации, обозначенного стереотипом «bind» (связать); это называют явным связыванием. Чтобы создать экземпляр шаблона, необходимо задать фактические значения, которые будут при-

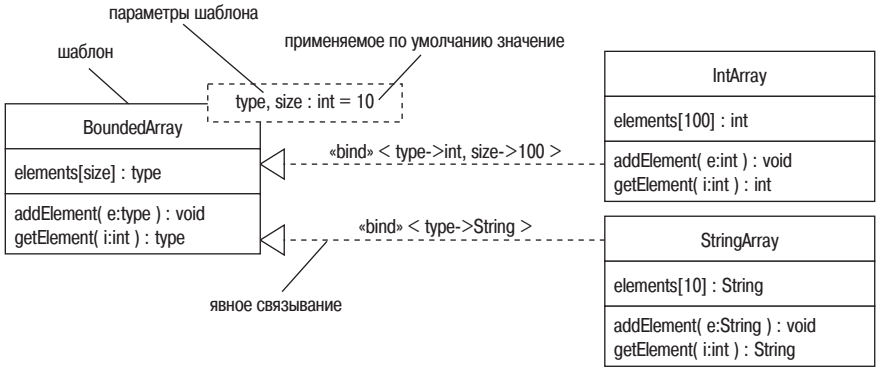


Рис. 17.10. *Создание экземпляра шаблона с помощью явного связывания*

своены параметрам шаблона. Их указывают в угловых скобках после стереотипа «bind». При создании экземпляра шаблона его параметры будут заменены этими значениями, в результате получится новый класс. Обратите внимание на синтаксис связывания параметров: выражение type->int означает, что параметр шаблона type при создании экземпляра замещается int. Символ -> можно прочитать как «замещается» или «связывается».

Очевидно, что шаблоны являются мощным механизмом многократного использования кода. Класс можно описать как шаблон, в общих чертах, и затем создавать множество его специальных версий, присваивая параметрам шаблона соответствующие реальные значения.

На рис. 17.10 связывание на самом деле используется двумя способами. Во-первых, классификатор связывается с параметром type. Если у параметра шаблона нет типа, то по умолчанию это классификатор. Во-вторых, параметру size присваивается реальное целое значение. Это позволяет задавать размер ограниченного массива как параметр шаблона.

В рамках конкретного шаблона имена параметров являются локальными. Это означает, что если у двух шаблонов есть параметр type, в каждом случае это разные параметры type.

Существует разновидность синтаксиса шаблонов, которую называют неявным связыванием. Здесь для представления создания экземпляра шаблона не используется явная реализация «bind», но происходит неявное связывание за счет применения специального синтаксиса в получаемых классах. Чтобы неявно создать экземпляр шаблона, фактические значения перечисляют в угловых скобках после имени класса шаблона, как показано на рис. 17.11. Недостаток этого подхода в том, что получаемому в результате классу нельзя присвоить имя.

По нашему мнению, лучше использовать явное связывание, чтобы классы, получаемые в результате создания экземпляров шаблона, могли иметь собственные описательные имена.

Хотя шаблоны – очень мощная возможность, в настоящее время C++ и Java являются единственными распространенными ОО языками программирования, которые их поддерживают (табл. 17.1). Понятно, что шаблоны могут использоваться при проектировании только в том случае, если они поддерживаются языком реализации.

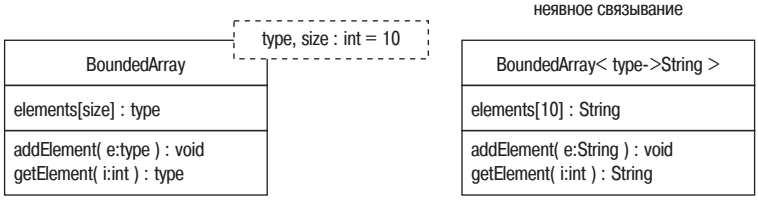


Рис. 17.11. *Создание экземпляра шаблона посредством неявного связывания*

Таблица 17.1

Язык программирования	Поддержка шаблонов
Java	Да
C++	Да
Smalltalk	Нет
Python	Нет
Visual Basic	Нет
C#	Нет

17.8. Вложенные классы

> Вложенный класс – это класс, определенный внутри другого класса.

Некоторые языки программирования, такие как Java, позволяют размещать описание класса внутри описания другого класса. Таким образом, создается так называемый вложенный класс. В Java их также называют внутренними классами (inner class).

Вложенный класс объявляется в пространстве имен его внешнего класса и доступен *только* для этого класса или объектов этого класса. Только внешний класс или его объекты могут создавать и использовать экземпляры вложенного класса.

Вложенные классы, как правило, относятся к вопросам проектирования. Здесь речь идет о том, *как* может быть реализована некоторая функциональность, а не *что* она из себя представляет.

Например, вложенные классы широко используются в Java при обработке событий. Пример на рис. 17.12 показывает простой класс окна HelloFrame (окно приветствия). Базовое поведение окна он наследует от своего родительского класса Frame. В HelloFrame есть вложенный класс

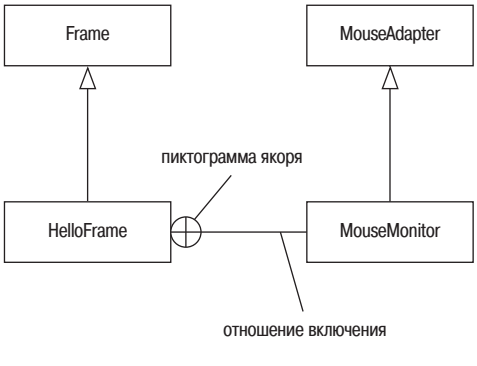


Рис. 17.12. *Класс MouseMonitor вложен в класс HelloFrame*

MouseMonitor (монитор мыши), который от своего родителя MouseAdapter наследует способность обрабатывать события мыши.

Каждый экземпляр HelloFrame использует экземпляр MouseMonitor для обработки событий мыши. Для этого экземпляр HelloFrame должен:

- создать экземпляр класса MouseMonitor;
- определить этот экземпляр MouseMonitor слушателем событий мыши.

Такой подход является хорошим стилем проектирования. Код обработки событий мыши выносится из остального кода HelloFrame в отдельный класс MouseMonitor. И MouseMonitor полностью инкапсулирован в HelloFrame.

17.9. Что мы узнали

Проектные классы – это строительные блоки проектной модели. Мы изучили следующее:

- Проектные классы разрабатываются в деятельности UP Проектирование класса.
- Проектные классы – это классы, описанные настолько полно, что могут быть реализованы.
- Существует два источника проектных классов:
  - предметная область:
    - уточнение классов анализа;
    - один класс анализа может превратиться в нуль, один или более проектных классов;
  - область решения:
    - библиотеки утилитных классов;
    - промежуточное программное обеспечение;
    - библиотеки GUI;
    - многократно используемые компоненты;
    - детали, характерные для конкретной реализации.
- Проектные классы имеют полные описания:
  - полный набор атрибутов, включающий:
    - имя;
    - тип;
    - применяемое по умолчанию значение, когда это необходимо;
    - видимость;
  - операций:
    - имя;
    - имена и типы всех параметров;
    - необязательные значения параметров, если это необходимо;

    - возвращаемый тип;
    - видимость.
- Правильно сформированные проектные классы имеют следующие характеристики:
    - открытые операции класса определяют контракт с клиентами класса;
    - полнота – класс делает не менее того, что его клиенты могут обоснованно ожидать;
    - достаточность – класс делает не более того, что его клиенты могут обоснованно ожидать;
    - простота – сервисы должны быть простыми, элементарными и уникальными;
    - высокая внутренняя связность:
        - каждый класс должен реализовывать единственное четко определенное абстрактное понятие;
        - все операции должны быть направлены на реализацию предназначения класса;
    - низкая связанность с другими классами:
        - класс должен быть взаимосвязан с таким количеством классов, которого достаточно для реализации его обязанностей;
        - между двумя классами должно устанавливаться взаимоотношение, только если между ними существует настоящая семантическая связь;
        - необходимо избегать использования связанности только с целью повторного использования некоего кода.
    - Проектный класс всегда необходимо оценивать с точки зрения клиентов этого класса.
- Наследование.
    - Наследование должно использоваться только тогда, когда между двумя классами существует четкое отношение «является» или с целью повторного использования кода.
    - Недостатки:
        - это самая строгая из возможных форм связанности между двумя классами;
        - в рамках иерархии наследования инкапсуляция слаба, что приводит к проблеме «хрупкости базового класса» – изменения базового класса передаются вниз по иерархии;
        - является очень негибкой формой отношений в большинстве языков программирования – отношение устанавливается во время компиляции и остается неизменным во время выполнения.

    - Подклассы всегда должны представлять «разновидность», а не «исполняемую роль» – для представления «исполняемой роли» должна использоваться агрегация.
- Множественное наследование позволяет классам иметь более одного родителя.
    - Из всех широко используемых ОО языков программирования только C++ поддерживает множественное наследование.
    - Рекомендации к проектированию:
        - все родительские классы при множественном наследовании должны быть семантически не связанными;
        - между классом и всеми его родителями должно быть установлено отношение «является разновидностью»;
        - к классу и его родителям должен применяться принцип замещаемости;
        - у самих родительских классов не должно быть общих родителей;
        - используйте смешанные классы – простые классы, разработанные для смешивания с другими классами при множественном наследовании; это надежное и мощное средство.
- Сравнение наследования и реализации интерфейса.
    - Наследование:
        - получаем интерфейс – открытые операции;
        - получаем реализацию – атрибуты, ассоциации, защищенные и закрытые члены.
    - Реализация интерфейса – получаем только интерфейс.
    - Наследование должно применяться, когда необходимо унаследовать некоторую реализацию.
    - Реализация интерфейса должна применяться, когда необходимо определить контракт.
- Шаблоны.
    - Из всех широко используемых ОО языков программирования в настоящее время шаблоны поддерживают только C++ и Java.
    - Шаблоны позволяют «параметризовать» тип – шаблон создается путем определения типа с помощью формальных параметров, и экземпляры шаблона создаются через связывание параметров с конкретными значениями.
        - Явное связывание использует зависимость, обозначенную стереотипом «bind»:
            - фактические значения показаны на отношении;
            - каждому экземпляру шаблона можно присвоить имя.

    - Неявное связывание:
        - фактические значения задаются в угловых скобках (< >) прямо в классе;
        - экземплярам шаблона имена присвоить нельзя – имена образуются именем шаблона и списком аргументов.
- Вложенные классы:
    - класс определяется внутри другого класса;
    - вложенный класс существует в пространстве имен внешнего класса – только внешний класс может создавать и использовать экземпляры вложенного класса;
    - в Java вложенные классы называются внутренними классами; их активно используют для обработки событий в классах GUI.

18

Уточнение отношений, выявленных при анализе

18.1. План главы

В этой главе обсуждаются методы уточнения (детализации) отношений, выявленных при анализе, в отношения уровня проектирования. Эти методы используются во всех деятельностях рабочего потока проектирования (рис. 16.6).

В первой части данной главы обсуждается преобразование отношений уровня анализа в одно из отношений целое-часть – агрегацию (раздел 18.4) или композицию (раздел 18.5).

Вторая часть посвящена работе с кратностью аналитических ассоциаций. Мы предлагаем конкретные методы уточнения аналитических ассоциаций с кратностью один-к-одному (раздел 18.7), многие-к-одному (раздел 18.8), один-ко-многим (раздел 18.9) и многие-ко-многим (раздел 18.11.1). Также рассматриваются двунаправленные ассоциации (раздел 18.11.2) и классы-ассоциации (раздел 18.11.3).

В завершение мы увидим, как с помощью UML 2 можно устанавливать отношение между составным классификатором и его частями.

18.2. Отношения уровня проектирования

> Аналитические ассоциации должны быть уточнены до отношений уровня проектирования, непосредственно реализуемых целевым ОО языком программирования.

При переходе к проектированию необходимо уточнить отношения между классами анализа и превратить их в отношения между проектными классами. Многие из выявленных при анализе отношений не могут

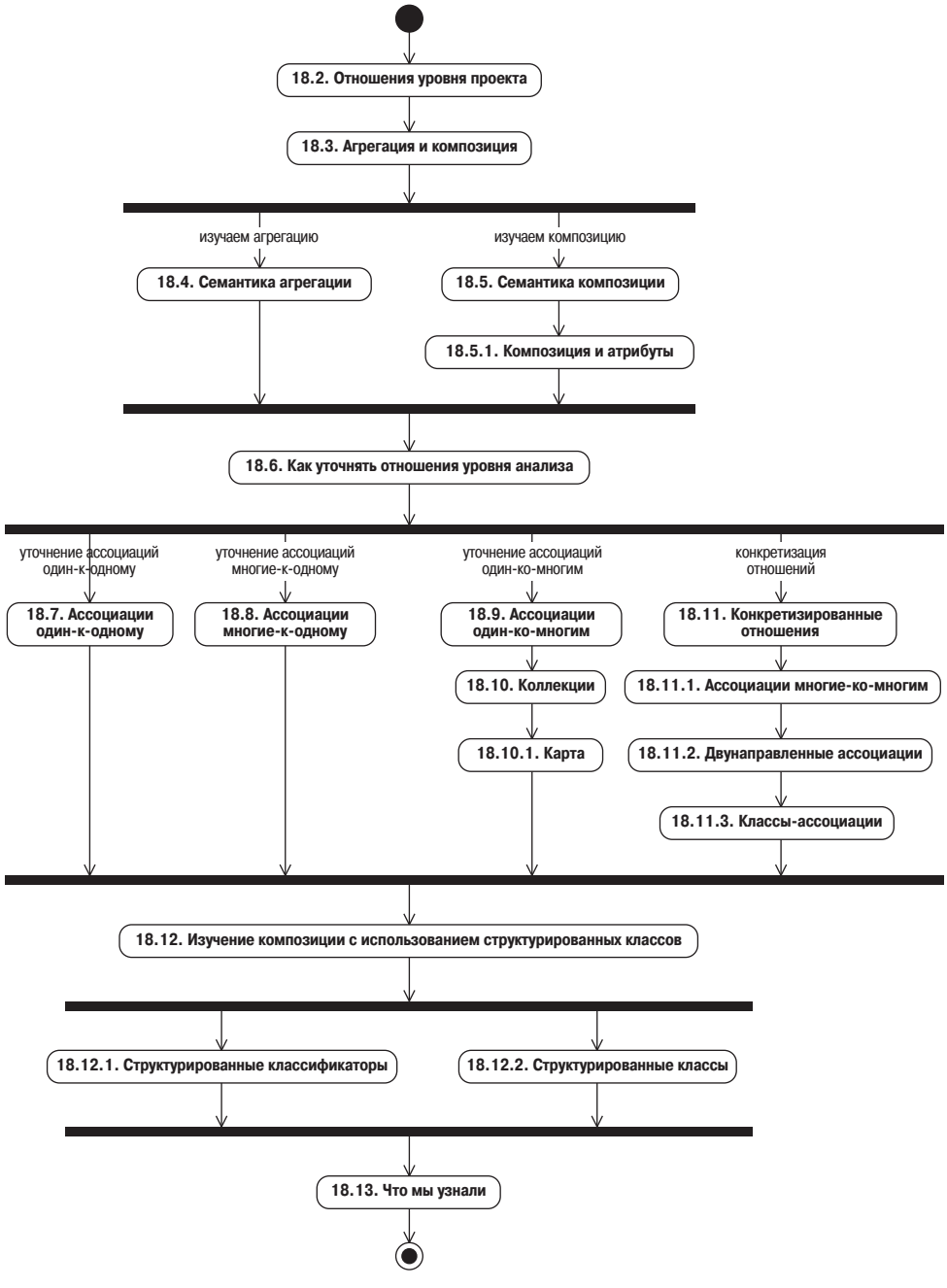


Рис. 18.1. *План главы*

быть реализованы как есть, их надо сделать таковыми. Например, ни один из широко распространенных ОО языков программирования не поддерживает двунаправленные ассоциации, классы-ассоциации или ассоциации многие-ко-многим. Чтобы создать проектную модель, необходимо определить, *как* должны быть реализованы эти ассоциации.

Уточнение аналитических ассоциаций до ассоциаций уровня проектирования включает несколько процедур:

- уточнение ассоциаций до отношений агрегации или композиции в соответствующих случаях;
- реализацию ассоциаций один-ко-многим;
- реализацию ассоциаций многие-к-одному;
- реализацию ассоциаций многие-ко-многим;
- реализацию двунаправленных ассоциаций;
- реализацию классов-ассоциаций.

Все ассоциации уровня проектирования *должны* обладать:

- возможностью навигации;
- кратностью на обоих концах.

У всех ассоциаций уровня проектирования *должно* быть указано имя ассоциации или имя роли, по крайней мере, на целевом конце.

18.3. Агрегация и композиция

При проектировании отношение ассоциация можно уточнить до агрегации или более строгой формы агрегации – композитной агрегации, если ассоциация имеет соответствующую семантику. Обычно композитную агрегацию называют просто композицией.

Получить представление о семантических отличиях между двумя типами агрегации можно из обсуждения примеров, взятых из окружающего нас мира.

- Агрегация – это свободный тип отношения между объектами – как между компьютером и его периферийным оборудованием.
- Композиция – это очень строгий тип отношения между объектами – как между деревом и его листьями.

Из примера, приведенного на рис. 18.2, можно сделать вывод, что компьютер очень слабо взаимосвязан со своим периферийным оборудованием. Периферийные устройства могут появляться и исчезать, могут совместно использоваться с другими компьютерами. Они не «принадлежат» конкретному компьютеру. Это пример агрегации. Напротив, дерево тесно взаимосвязано со своими листьями. Листья принадлежат только одному дереву, ими нельзя поделиться с другими деревьями, и когда дерево умирает, листья умирают вместе с ним. Это пример композиции.

Рис. 18.2. *Два типа ассоциации в UML*

Очень полезно помнить об этих простых аналогиях при дальнейшем более подробном рассмотрении семантики агрегации и композиции.

18.4. Семантика агрегации

Агрегация – это тип отношения целое-часть, в котором агрегат образуется многими частями. В отношении целое-часть один объект (целое) использует сервисы другого объекта (части). По существу, целое является доминантной и управляющей стороной отношения, тогда как часть просто обслуживает запросы целого и, следовательно, играет более пассивную роль. Действительно, если навигацию можно осуществлять только от целого к части, последняя даже не знает, что является частью целого.

Рассмотрим конкретный пример агрегации (рис. 18.3). Мы видим следующее:

- компьютер (класс Computer) может быть присоединен к 0 или более принтерам (класс Printer);
- в любой *момент времени* принтер соединен с 0 или 1 компьютером;
- в *ходе времени* конкретный принтер может использоваться многими компьютерами;
- принтер может существовать, даже если не подключен ни к одному компьютеру;
- принтер фактически не зависит от компьютера.

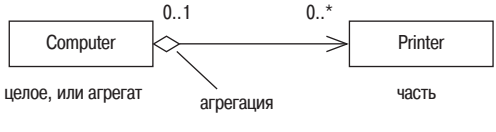
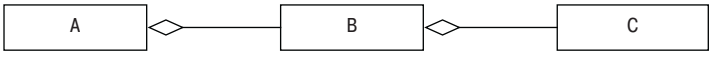


Рис. 18.3. *Пример агрегации*

> Агрегация – это отношение целое-часть.

Семантику агрегации можно подытожить следующим образом:

- агрегат может существовать как независимо от частей, так и вместе с ними;
- части могут существовать независимо от агрегата;
- агрегат является в некотором смысле неполным в случае отсутствия некоторых частей;
- части могут принадлежать одновременно нескольким агрегатам.



Агрегация транзитивна: если C является частью B, и B является частью A, тогда C также является частью A

Рис. 18.4. *Агрегация транзитивна*

Агрегация транзитивна. Рассмотрим пример на рис. 18.4. Транзитивность означает, что если C является частью B, и B является частью A, тогда C также является частью A.

> Агрегация транзитивна.

Агрегация асимметрична. Это означает, что объект никогда – ни прямо, ни косвенно – не может быть частью самого себя. Это ограничивает способы использования агрегации в моделях. В примере на рис. 18.5 показано, что объекты Product могут состоять из других объектов Product. В данном случае ошибки нет, потому что это *разные* объекты, и ограничение асимметрии не нарушено.

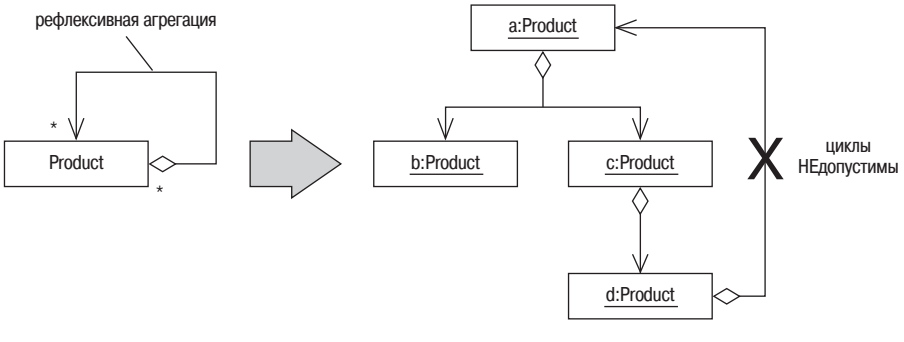


Рис. 18.5. *Ограничение асимметрии для агрегации не нарушено, т. к. здесь представлены разные объекты Product*

> Агрегация асимметрична. Объект никогда не может быть частью самого себя.

Продолжим рассмотрение рис. 18.5. Иногда может понадобиться смоделировать ситуацию, когда между объектами d и a существует связь, как показано на рисунке. Такое может случиться, если объекту d необходимо осуществить *обратный вызов* и использовать некоторые сервисы объекта-агрегата a. Но как это смоделировать на диаграмме классов? Рефлексивная агрегация (reflexive aggregation) с классом Product не подходит, потому что согласно ограничению асимметрии для агрегации объект a не может быть частью самого себя ни прямо, ни косвенно. Поэтому для обработки связи между объектами d и a необходимо использовать рефлексивную неуточненную ассоциацию класса Product с самим собой, как показано на рис. 18.6.

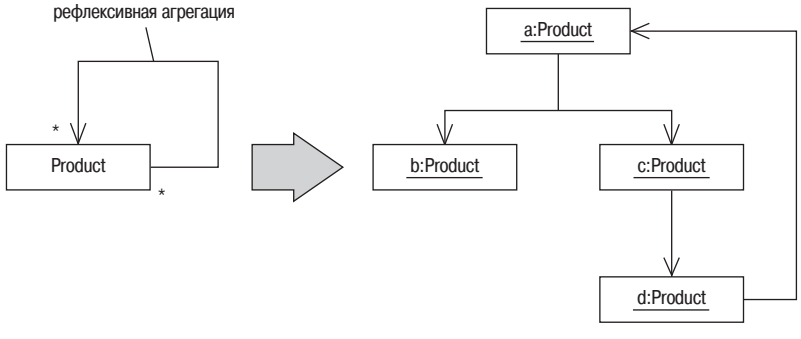


Рис. 18.6. *Связь между объектами a и d реализована посредством рефлексивной неуточненной ассоциации*

> Ассоциация симметрична. Объект может быть ассоциирован с самим собой.

На рис. 18.7 показан другой типичный пример агрегации. Домашний компьютер (целое) может быть смоделирован как набор частей. Эти части довольно слабо взаимосвязаны с целым. Их можно переносить с компьютера на компьютер или использовать совместно с другими компьютерами, поэтому в этой модели применима семантика агрегации. Согласно этой модели домашний компьютер можно рассматривать как совокупность следующих частей: Mouse (мышь), Keyboard (клавиатура), CPU (центральный процессор), Monitor (монитор) и два объекта Speaker (колонка). CPU, в свою очередь, может быть смоделирован как совокупность различных аппаратных компонентов, таких как RAM (ОЗУ), HardDrive (жесткий диск) и т. д.

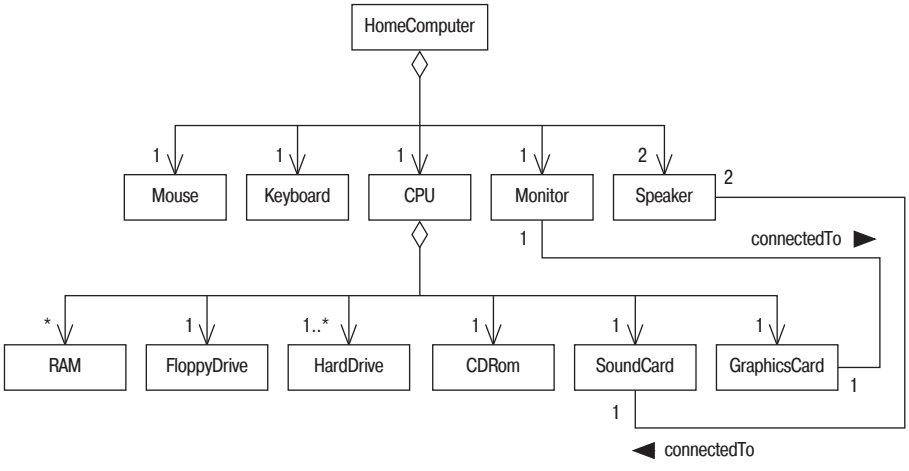


***Рис. 18.7.** Типичный пример агрегации: компьютер как совокупность его частей*

18.5. Семантика композиции

Композиция – более строгая форма агрегации. Она имеет сходную (но более ограниченную) семантику. Как и агрегация, это отношение целое-часть, являющееся как транзитивным, так и асимметричным.

Ключевое различие между агрегацией и композицией в том, что в композиции у частей *нет* независимой жизни вне целого. Более того, в композиции каждая часть принадлежит максимум одному и только одному целому, тогда как при агрегации часть может совместно использоваться несколькими целыми.

> Композиция – это строгая форма агрегации.

В примере на рис. 18.8 объекты Button (кнопка) не могут существовать независимо от владеющего ими объекта Mouse. Если уничтожается объект Mouse, уничтожаются и принадлежащие ему объекты Button, потому что они являются его неотъемлемой частью. Каждый объект Button может принадлежать только одному объекту Mouse. Так же и листья на деревьях – жизнь листа определяется жизнью дерева, и лист может принадлежать только одному дереву.

***Рис. 18.8.** Пример композиции*

> Композит имеет исключительное право владения и ответственности за свои части.

Резюмировать семантику композиции можно следующим образом:

- одновременно части могут принадлежать только одному композиту – совместное владение частями невозможно;
- композит обладает исключительной ответственностью за все свои части; это означает, что он отвечает за их создание и уничтожение;
- композит может высвобождать части, передавая ответственность за них другому объекту;
- в случае уничтожения композита он должен или уничтожить все свои части, или передать ответственность за них другому объекту.

Поскольку композит обладает исключительной ответственностью за жизненный цикл и управление своими частями, то при создании объект композита часто создает и свои части. Аналогично при уничтожении композита он должен уничтожить все свои части *или* организовать их передачу другому композиту.

Несмотря на то, что существуют *и* иерархии, *и* сети рефлексивной агрегации, для рефлексивной композиции возможны только иерархии. В этом состоит еще одно отличие между агрегацией и композицией, которое объясняется тем, что в композиции объект-часть в любой момент времени может быть частью только *одного* композита.

18.5.1. Композиция и атрибуты

> Часть композита эквивалентна атрибуту.

Рассматривая семантику композиции, можно заметить, что она очень похожа на семантику атрибутов. Жизненный цикл в обоих случаях контролируется владельцами. И части, и атрибуты не могут независимо существовать вне своих владельцев. В сущности, атрибуты – это точный эквивалент отношения композиции между классом композита и классом атрибута. Тогда зачем нужны два способа описания одного и того же? На это есть две причины:

- Типом атрибута может быть простой тип данных. В некоторых гибридных ОО языках программирования, таких как C++ и Java, есть простые типы, например int и double, которые не являются классами. Их можно было бы моделировать как классы, обозначенные стереотипом «primitive», но это загромождало бы модель. Простые типы *всегда* должны моделироваться как атрибуты.
- Существуют определенные, широко используемые утилитные классы, такие как Time, Date и String. Если бы каждый раз приходилось моделировать эти классы с помощью отношения композиции класса

с самим собой, очень скоро модели стали бы абсолютно непонятными. Намного лучше моделировать такие классы как атрибуты.

Вывод состоит в следующем: если имеется простой тип, или утилитный класс, или даже класс, который не стоит явно показывать в модели, поскольку он не представляет особого интереса либо от него мало пользы, необходимо рассмотреть возможность применения атрибута, а не отношения композиции. Здесь нет никакого конкретного и твердого правила, но главное, о чем надо всегда помнить, – это понятность, полезность и удобочитаемость модели.

18.6. Как уточнять отношения уровня анализа

При анализе использовались простые ассоциации без какого-либо подробного рассмотрения семантики отношения (или того, каким образом отношение должно быть реализовано). Однако при проектировании необходимо всегда пытаться достичь максимальной конкретизации и уточнить ассоциации до одного из отношений агрегации везде, где это возможно. Фактически ассоциация *должна* использоваться в проектировании, только если в противном случае в схеме агрегации образуется цикл (см. раздел 18.4). Такая ситуация встречается редко, поэтому большинство аналитических ассоциаций превращаются или в агрегацию, или в композицию.

> Аналитические ассоциации должны быть уточнены до одного из отношений агрегации везде, где это возможно.

После того как принято решение о применении агрегации или композиции, необходимо действовать следующим образом:

- добавить в ассоциации кратности и имена ролей, если они отсутствуют;
- выбрать, какой конец ассоциации является целым, а какой – частью;
- посмотреть на кратность связи со стороны целого; если это 0..1 или 1, *вероятно*, можно использовать композицию; в противном случае *должна* использоваться агрегация;
- добавить возможность навигации *от* целого *к* части – ассоциации уровня проектирования должны быть однонаправленными.

Таким образом, происходит уточнение ассоциации до агрегации или композиции.

Если кратности целого или части больше 1, необходимо принять решение, как это будет реализовываться. Это второй шаг уточнения.

18.7. Ассоциации один-к-одному

Практически всегда ассоциация один-к-одному превращается в композицию. По сути, ассоциация один-к-одному подразумевает настолько строгое отношение между двумя классами, что зачастую стоит рассмотреть возможность их объединения в один класс без нарушения правил проектирования для проектных классов (раздел 17.5). Если классы нельзя объединить, отношение один-к-одному уточняется до композиции, как показано на рис. 18.9.

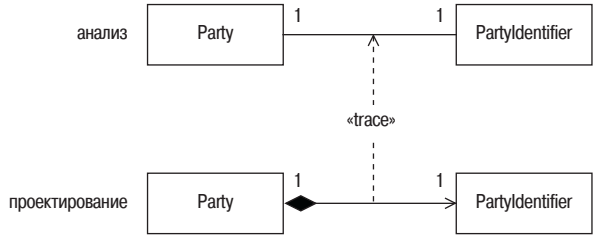


Рис. 18.9. *Ассоциация один-к-одному уточняется до композиции*

> Обычно ассоциации один-к-одному подразумевают композицию.

Также можно было бы рассмотреть возможность превращения PartyIdentifier (идентификатор партии) в атрибут класса Party (партия), если PartyIdentifier не является особо важным классом. Это, конечно, упрощает диаграмму (рис. 18.10), но имеет и недостаток: нельзя показать атрибуты или операции, принадлежащие классу PartyIdentifier.

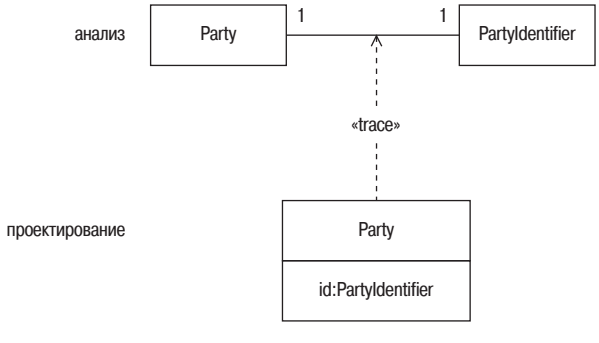


Рис. 18.10. *Класс PartyIdentifier превращен в атрибут класса Party*

18.8. Ассоциации многие-к-одному

В ассоциации многие-к-одному целое имеет кратность «много», а кратность части равна 1.

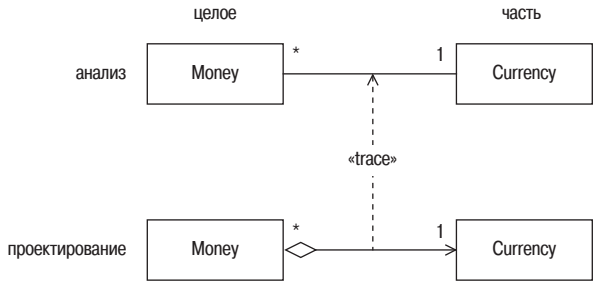


Рис. 18.11. *Ассоциация многие-к-одному уточняется до агрегации*

Поскольку со стороны целого кратность равна «много», сразу понятно, что композиции *невозможна*, потому что часть используется совместно многими целыми. Но, *вероятно*, возможна агрегация. Здесь необходимо провести проверку на наличие циклов в схеме агрегации (см. раздел 18.4). Если таковых не обнаружено, аналитическую ассоциацию можно уточнять до агрегации, как показано на рис. 18.11. Как видно из этого примера, один объект Currency (валюта) совместно используется многими объектами Money (деньги). Это абсолютно верно отражает отношение между деньгами и валютой: деньги – это сумма в некоторой валюте. Намного более полную модель денег можно найти в книге [Arlow 1].

> Ассоциации многие-к-одному подразумевают агрегацию, если в схеме агрегации нет цикла.

18.9. Ассоциации один-ко-многим

В ассоциации один-ко-многим со стороны части присутствует *коллекция* объектов. Чтобы реализовать такое отношение, необходимо использовать или поддержку коллекций, предоставляемую языком реализации, или классы-коллекции.

Большинство ОО языков программирования имеют минимальную встроенную поддержку коллекций объектов. В сущности, большинство языков предлагают только массивы. Массив – это индексированная коллекция объектных ссылок, обычно ограниченная некоторым максимальным размером. Преимущество встроенных массивов, как правило, состоит в их высокой производительности. Однако это преимущество нивелируется их малой гибкостью в сравнении с другими типами коллекций.

Классы-коллекции обычно характеризуются намного большей мощью и гибкостью, чем массивы. Они предлагают разнообразные семантики, в которых массив является лишь одним из возможных вариантов. Далее вся глава посвящена классам-коллекций.

18.10. Коллекции

Класс-коллекции – это класс, экземпляры которого специализируются на управлении коллекциями других объектов. В большинстве языков программирования есть стандартные библиотеки классов-коллекций (и других утилит).

> Классы-коллекции – это классы, экземпляры которых содержат коллекции объектов.

Одним из ключей к отличному ОО проектированию и реализации является совершенное владение классами-коллекций. У всех таких классов есть операции для:

- добавления объектов в коллекцию;
- удаления объектов из коллекции;
- извлечения ссылки на объект, находящийся в коллекции;
- обход коллекции, т. е. проход по коллекции от первого объекта до последнего.

Существует много типов коллекций, по-разному обрабатывающих коллекции объектов. Важным аспектом ОО проектирования и реализации является правильный выбор типа коллекции (обсуждается далее).

В качестве примера использования коллекций на рис. 18.12 показана ассоциация один-ко-многим уровня анализа, реализованная с помощью класса-коллекции Vector. Это класс стандартной библиотеки Java java.util. Как правило, между целым (классом Order (заказ)) и классом Vector устанавливается отношение композиции, поскольку Vector обычно является всего лишь частью реализации целого и не может существовать вне него. Однако между классом Vector и его частями (OrderLine (строка заказа)) может существовать отношение агрегации или композиции. Если целое несет ответственность за жизненный цикл частей, как в данном примере, можно использовать композицию. В противном случае должна использоваться агрегация.

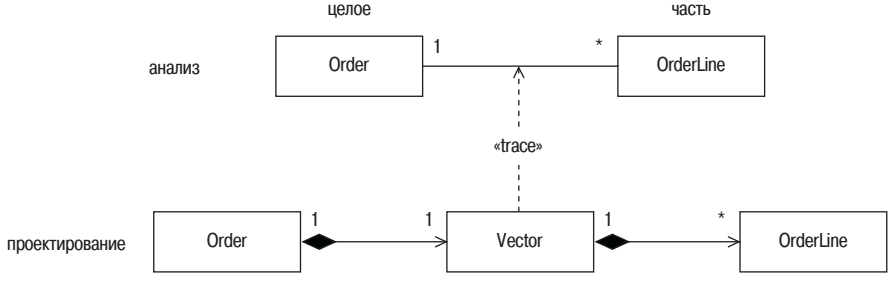


Рис. 18.12. *Реализация ассоциации один-ко-многим с помощью класса-коллекции Vector*

Классы-коллекции реализуют ассоциации один-ко-многим.

С точки зрения моделирования с использованием коллекций существует четыре фундаментальные стратегии:

- Класс-коллекции моделируется явно; этот вариант представлен на рис. 18.12. Его преимущество в чрезвычайной детализации, однако есть и большой недостаток – загроможденность проектной модели. Если ассоциация один-ко-многим заменяется классом-коллекции, модель быстро становится очень раздутой. Выбор класса-коллекции обычно является тактическим решением реализации, и право это сделать можно передать программистам. Мы рекомендуем заменять ассоциации один-ко-многим конкретными классами-коллекций, *только* если выбор коллекции является стратегически важным.
- Инструментальному средству моделирования сообщается о том, как будет реализовываться каждая конкретная ассоциация один-ко-многим. Многие инструменты моделирования, генерирующие код, позволяют назначать конкретный класс-коллекции для каждой ассоциации один-ко-многим. Обычно это осуществляется путем добавления в ассоциацию помеченных значений, определяющих свойства генерации кода для данного отношения. Этот подход отображен на рис. 18.13. Здесь на соответствующем конце отношения указано свойство {Vector}. Обратите внимание, что используется только именная часть помеченного значения – часть «значение» в данном случае является излишней.

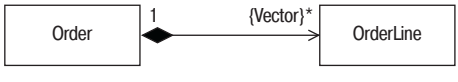


Рис. 18.13. *Добавление в ассоциацию помеченного значения*

Опасайтесь «переусердствовать» при использовании коллекций.

- Путем добавления свойства к отношению определяется семантика коллекции, но не указывается какой-либо конкретный класс реализации (рис. 18.14). При использовании коллекций важно не «переусердствовать». Как говорилось, фактически используемый тип коллекции – часто тактическое, а не стратегическое решение, которое может быть принято программистами во время реализации.

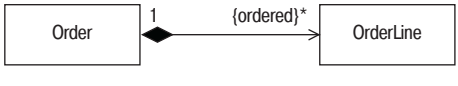


Рис. 18.14. *Добавление в ассоциацию свойства без указания конкретного класса реализации*

Этот вариант хорош тем, что сохраняется лаконичность модели, и программистам могут быть предоставлены подсказки по поводу того, какой класс-коллекции должен использоваться. Однако такой подход обычно исключает автоматическую генерацию кода.

- Уточнением отношения один-ко-многим до классов-коллекций не занимаются – решение этой задачи передается программистам.

UML предоставляет стандартные свойства, которые могут применяться к кратностям для обозначения требуемой семантики коллекции. Они представлены в табл. 18.1.

Упорядоченность определяет, как располагаются элементы в коллекции: в строгом порядке относительно друг друга или нет. Уникальность определяет единичность каждого объекта коллекции. Для отношения один-ко-многим по умолчанию применяется семантика {unordered, unique} ({неупорядоченный, уникальный}).

Таблица 18.1

Стандартное свойство	Семантика
{ordered}	Существует строгий порядок расположения элементов коллекции.
{unordered}	Элементы в коллекции располагаются в произвольном порядке.
{unique}	Все элементы коллекции уникальны – один и тот же объект появляется в коллекции максимум один раз.
{nonunique}	В коллекции допускается дублирование элементов.

Например, если необходима упорядоченная коллекция объектов OrderLines, то изображенное на рис. 18.13 можно показать, как представлено на рис. 18.14.

Сочетания свойств упорядочения и уникальности создают набор коллекций, перечисленных в табл. 18.2. Эти коллекции являются частью OCL (объектный язык ограничений) (см. главу 25), хотя аналогичный набор коллекций есть во всех языках программирования.

Таблица 18.2

Свойство	Коллекция OCL
{unordered, nonunique}	Bag (мультимножество)
{unordered, unique}	Set (множество)
{ordered, unique}	OrderedSet (упорядоченное множество)
{ordered, nonunique}	Sequence (последовательность)

18.10.1. Карта

> Карты оптимизированы с целью быстрого получения значения по ключу.

Еще один очень полезный тип класса коллекции – карта (map). Иногда его называют словарем (dictionary).[1] Эти классы немного похожи на таблицу базы данных с двумя столбцами – ключ и значение. Карты спроектированы таким образом, чтобы по заданному ключу можно было *быстро* найти соответствующее значение. Если необходимо хранить коллекции объектов, доступ к которым осуществляется по значению уникального ключа, или необходимо создать индексы быстрого доступа в другие коллекции, карта – оптимальное решение.

Обычно работа карты заключается в обслуживании набора узлов, где каждый узел указывает на два объекта – объект-ключ и объект-значение. Они оптимизированы так, чтобы быстро находить объект-значение по заданному объекту-ключу.

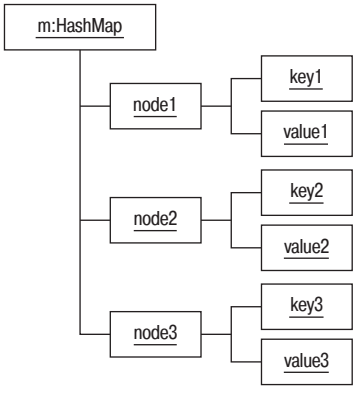


Рис. 18.15*. Пример карты*

На рис. 18.15 показано упрощенное представление Java-класса HashMap. Поиск значения (по заданному ключу) осуществляется очень быстро, поскольку коллекция проиндексирована с использованием хеш-таблицы.

В UML нет стандартного свойства для обозначения карты, и карты не являются частью OCL. Если в проектной модели необходимо обозначить применение карты, указывается конкретный тип коллекции (например, {HashMap}) или используется следующее помеченное значение:

```
{map имяКлюча}
```

Если принято решение использовать это нестандартное обозначение, в модель необходимо добавить поясняющее примечание!

---

[1] Еще известно как «ассоциативный массив». – *Примеч. науч. ред.*

18.11. Конкретизированные отношения

Некоторые типы отношений являются исключительно артефактами анализа и не поддерживаются напрямую ни одним из широко используемых ОО языков программирования. Процесс реализации таких отношений уровня анализа называют *конкретизацией* (*reification*) (т. е. превращение в нечто конкретное). Конкретизации подвергаются следующие отношения уровня анализа:

- ассоциации многие-ко-многим;
- двунаправленные ассоциации;
- классы-ассоциации.

18.11.1. Ассоциации многие-ко-многим

Ни один из широко используемых ОО языков программирования не поддерживает напрямую ассоциации многие-ко-многим (хотя некоторые объектные базы данных все-таки их поддерживают). Поэтому эти ассоциации необходимо конкретизировать в обычные классы, отношения агрегации, композиции и зависимости. При анализе такие вопросы, как владение и навигация, могли бы оставаться неясными, но в проектировании такая неопределенность невозможна. Это тот случай, когда сначала надо решить, какая из сторон ассоциации многие-ко-многим является целым, а затем применять соответственно агрегацию или композицию.

В примере на рис. 18.16 отношение allocation (распределение) было конкретизировано в класс Allocation. Это превращает ассоциацию многие-ко-многим в два отношения агрегации. Исходя из требований, предъявляемых к системе, было принято решение, что Resource (ресурс) – это целое. Система главным образом занимается управлением потоками работ, ассоциированными с Resource, т. е. является ресурсоцентричной. Однако если бы система была задаче-центричной, целым был бы сделан объект Task (задача), что изменило бы направление отношений, представленных на рисунке.

Было также решено возложить ответственность за жизненный цикл объектов Allocation на Resource, и поэтому используется отношение композиции.

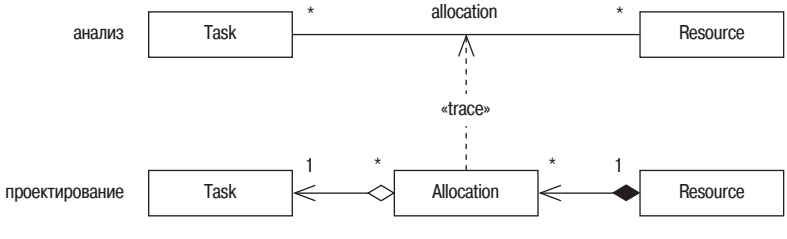


Рис. 18.16. *Ассоциация многие-ко-многим конкретизирована в класс Allocation*

Если система пытается представить *обе* точки зрения, ее можно было бы назвать ориентированной на распределение ресурсов. Тогда был бы введен новый объект (возможно, AllocationManager (менеджер распределения)), обслуживающий список объектов Allocation, каждый объект которого указывает как на объект Resource, так и на объект Task.

18.11.2. Двунаправленные ассоциации

Часто необходимо смоделировать обстоятельства, в которых объект a класса A использует сервисы объекта b класса B, и объекту b необходимо делать обратный вызов и использовать сервисы объекта a. В качестве примера можно привести элемент управления GUI Window с одним или более объектами Button, где каждому объекту Button надо выполнять обратный вызов и использовать сервисы Window, которому они принадлежат.

При анализе все просто – такая ситуация моделируется как единственная двунаправленная ассоциация. Однако ни один из широко используемых ОО языков программирования реально не поддерживает двунаправленные ассоциации. Поэтому при проектировании такая ассоциация должна быть конкретизирована в две однонаправленные ассоциации или зависимости, как показано на рис. 18.17.

При моделировании обратных вызовов необходимо помнить об асимметрии агрегации и композиции – объект *никогда* – ни прямо, ни косвенно – не может быть частью самого себя. Это значит, что если класс A имеет отношение агрегации или композиции с классом B, обратный вызов от B к A должен быть смоделирован как неконкретизированная ассоциация. Если бы между B и A было применено отношение агрегации, объект b был бы частью (через композицию или агрегацию) объекта a, и объект a был бы частью (через агрегацию) объекта b. Такое замкнутое владение, конечно же, нарушает асимметричность агрегации.

Двунаправленные ассоциации также имеют место тогда, когда целое передает ссылку на себя в качестве параметра одной из операций части или когда часть в одной из своих операций создает экземпляр целого. В этих случаях необходимо использовать отношение зависимости от части к целому, а не ассоциацию.

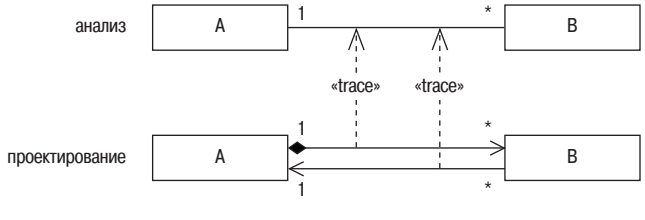


Рис. 18.17. *При проектировании двунаправленная ассоциация конкретизируется в две однонаправленные ассоциации*

18.11.3. Классы-ассоциации

Классы-ассоциации – исключительно аналитические артефакты, которые напрямую не поддерживаются ни одним из широко используемых ОО языков программирования. Таким образом, они не имеют прямого аналога в проектировании и должны быть удалены из проектной модели.

Класс-ассоциация конкретизируется в обычный класс, а для отражения его семантики используется сочетание ассоциации, агрегации, композиции или даже зависимости. Это может потребовать наложения ограничений на модель. Принимается решение, какая из сторон ассоциации является целым, и соответственно этому используется композиция, агрегация и возможность навигации. Пример показан на рис. 18.18.

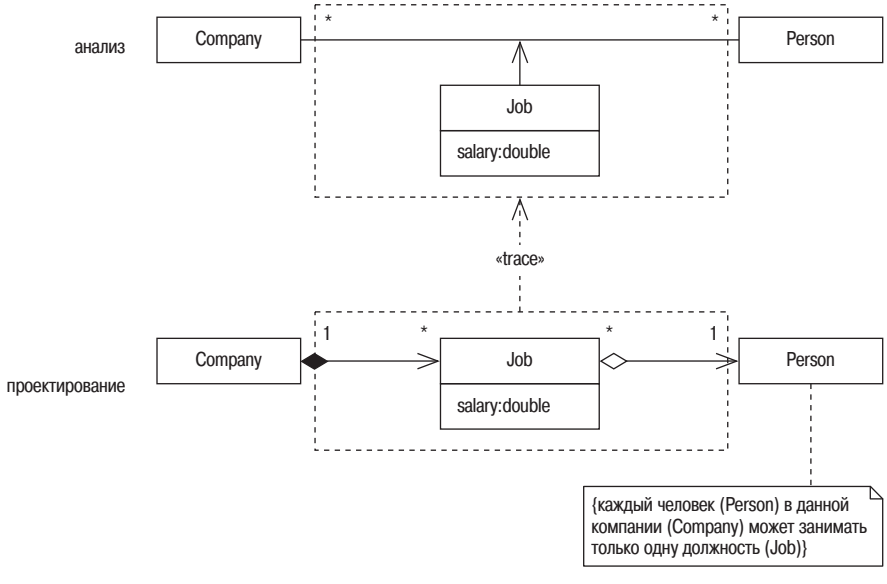


Рис. 18.18. *Пример конкретизации класса-ассоциации*

Обратите внимание, что при конкретизации класса-ассоциации теряется его семантика, которая гласит, что объекты на каждом конце класса-ассоциации должны формировать уникальную пару (см. раздел 9.4.5). Однако, как показано на рис. 18.18, добавив примечание, содержащее соответствующее ограничение, эту семантику можно без труда восстановить.

18.12. Изучение композиции с использованием структурированных классов

До сих пор мы брали отношения, выявленные при анализе, и превращали их в одно или более отношений уровня проектирования. В этом состоит основная деятельность в уточнении отношений уровня анализа. Однако UML 2 также позволяет анализировать отношение между составным классификатором и его частями. Это может быть важной частью деятельностей UP Проектирование класса, Проектирование прецедента и Проектирование подсистемы, поскольку позволяет сосредоточить внимание на внутренних действиях одного из этих классификаторов. Основным понятием здесь является структурированный классификатор, который рассматривается в следующем разделе.

18.12.1. Структурированные классификаторы

> Структурированный классификатор – это классификатор, имеющий внутреннюю структуру.

Структурированный классификатор (structured classifier) – это просто классификатор (такой как класс), имеющий внутреннюю структуру. Эта структура моделируется как части, объединенные с помощью соединителей. Взаимодействие структурированного классификатора с его окружением моделируется его интерфейсами и портами, но мы отложим их обсуждение до главы 19.

Часть – это роль, которую могут выполнять один или более экземпляров классификатора в контексте структурированного классификатора. Каждая часть может иметь:

- имя роли – описательное имя роли, исполняемой экземплярами в контексте структурированного классификатора;
- тип – только экземпляры этого типа (или подтипа этого типа) могут играть эту роль;
- кратность – число экземпляров, которые могут играть роль в любой конкретный момент времени.

> Соединители – это отношения между ролями (частями). Соединители и части существуют только в рамках контекста конкретного структурированного классификатора.

Соединитель (connector) – это отношение между частями в контексте структурированного классификатора. Он указывает на то, что части могут общаться друг с другом и что между экземплярами, играющими роль частей, через которые это общение может происходить, существует отношение. Эти отношения могут проецироваться в ассоциации между классами частей. Или они могут быть просто специальными от-

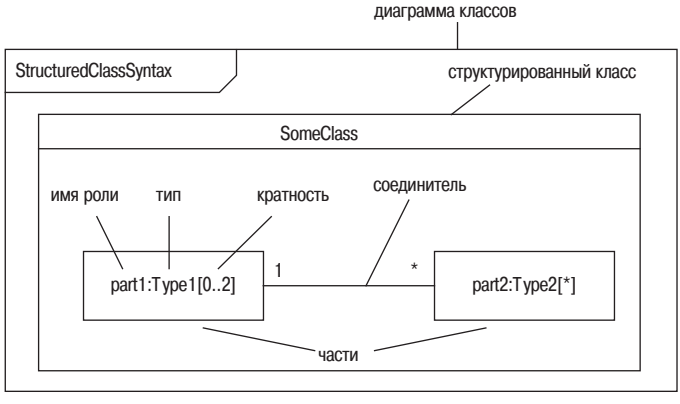


Рис. 18.19. *Синтаксис структурированного классификатора*

ношениями, при которых структурированный классификатор объединяет части во временную кооперацию для осуществления некоторой задачи.

Синтаксис структурированного классификатора показан на рис. 18.19 на примере класса SomeClass (некоторый класс).

Основные моменты синтаксиса структурированного классификатора:

- части кооперируются в контексте структурированного классификатора;
- части представляют роли, которые могут играть экземпляры классификатора в контексте структурированного классификатора – части *не* представляют классы;
- соединитель – это отношение между двумя частями, которое указывает на то, что экземпляры, играющие роли, определенные частями, могут общаться некоторым образом.

Из этого списка видно, что при моделировании структурированного классификатора учитываются только внутренняя реализация и внешний интерфейс одного классификатора. Все классификаторы, не являющиеся частями структурированного классификатора, игнорируются. Таким образом, это очень узкоспециализированная техника моделирования.

18.12.2. Структурированные классы

Структурированный класс имеет дополнительное ограничение: он владеет всеми своими частями, соединителями и портами (см. раздел 19.6). Между ними и классом устанавливается неявное отношение включения.

Давайте рассмотрим пример использования структурированных классов на практике. На рис. 18.20 показана диаграмма классов для прос-

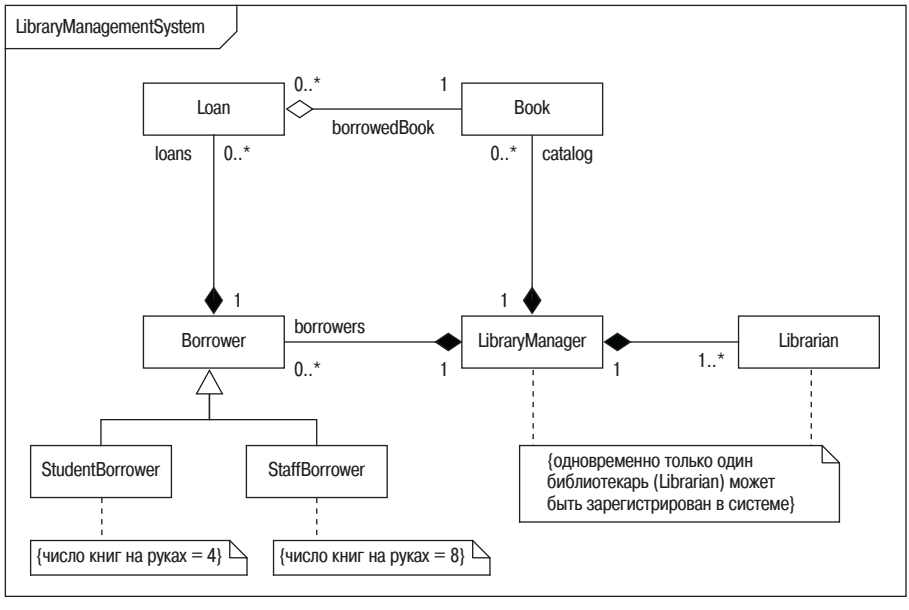


Рис. 18.20. *Диаграмма классов для системы управления библиотекой*

той системы управления библиотекой. На нее наложены следующие бизнес-ограничения:

- существует два типа класса Borrower (читатель) – StudentBorrower (читатель-студент) и StaffBorrower (читатель-сотрудник);
- у StudentBorrower одновременно на руках может быть максимум четыре книги;
- у StaffBorrower одновременно на руках может быть максимум восемь книг;
- одновременно в системе может быть зарегистрирован только один Librarian (библиотекарь) – это однопользовательская система.

Все атрибуты и операции на этом рисунке скрыты, потому что основное внимание направлено на аспекты структуры системы.

Как видите, у класса LibraryManager (менеджер библиотеки) есть внутренняя структура – сочетание Borrower, Book и Librarian. Из-за транзитивной природы композиции классы Loan (выдача) также являются частью этой составной структуры. Класс LibraryManager можно представить как структурированный класс, как показано на рис. 18.21. Обратите внимание, что классы StudentBorrower и StaffBorrower тоже отображены как структурированные. Кстати, структурированные классы могут быть вложенными с любым уровнем вложенности.

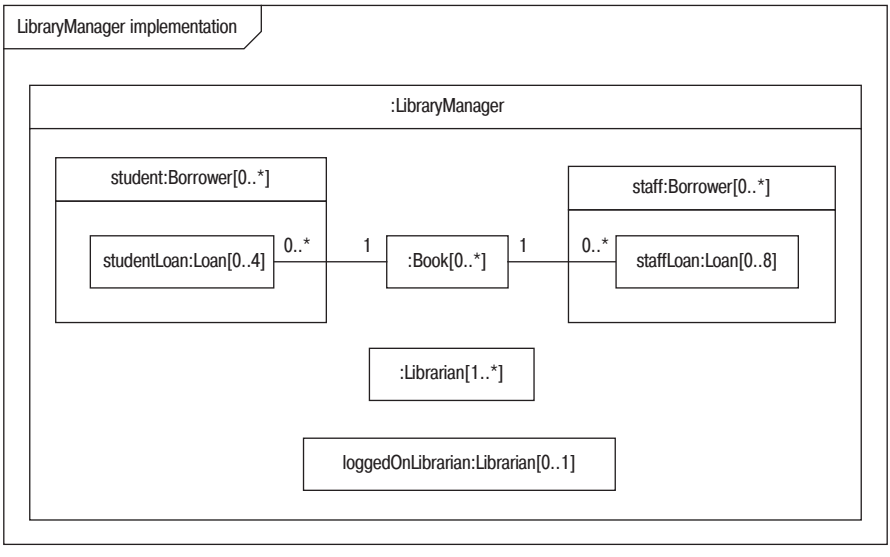


Рис. 18.21. *Структурированный класс LibraryManager*

Рисунок 18.21 дает несколько другое представление системы и позволяет взглянуть на класс LibraryManager немного подробнее, рассмотрев роли, исполняемые экземплярами класса в его реализации.

- С точки зрения менеджера библиотеки (LibraryManager) существует два типа читателей (Borrower), которые обрабатываются немного поразному исходя из максимального количества выданных книг (Loan). Чтобы представить это, вводятся новые роли – student (студент) и staff (штат).
- У student и staff может быть разное максимальное количество невозвращенных Loan в любой момент времени. Чтобы показать это, создаются две разные роли Loan – studentLoan и staffLoan – и для каждой из них вводится соответствующая кратность.
- LibraryManager допускает одновременную регистрацию в системе только одного Librarian, поэтому введена роль loggedOnLibrarian (зарегистрированный библиотекарь) с кратностью 0..1.

> Некоторые роли ассоциаций могут проецироваться в роли частей.

Как видите, мы смогли явно представить внутренние роли, исполняемые экземплярами в рамках контекста класса LibraryManager. Заметьте, что эти роли могут *отличаться* от ролей, исполняемых классами в их ассоциациях с LibraryManager. Например, на рис. 18.20 класс Borrower играет роль borrowers в своей ассоциации с LibraryManager. А на рис. 18.21 эта роль была уточнена и превратилась в более конкретные роли под-

классов Borrower, student и staff. Как правило, некоторые роли ассоциаций проецируются в роли частей, а некоторые – нет.

Аналогично соединители могут проецироваться в ассоциации между классами или только во временные отношения, созданные в контексте структурированного класса. В этом простом примере каждый соединитель может быть отображен в ассоциацию.

Внутреннюю структуру структурированных классификаторов можно представить как на диаграмме классов, так и на специальной диаграмме, которую называют *диаграммой составных структур* (*composite structure diagram*). Имя диаграммы соответствует имени структурированного классификатора, а содержимое диаграммы – это содержимое структурированного классификатора. На рис. 18.22 изображена диаграмма составных структур класса LibraryManager.

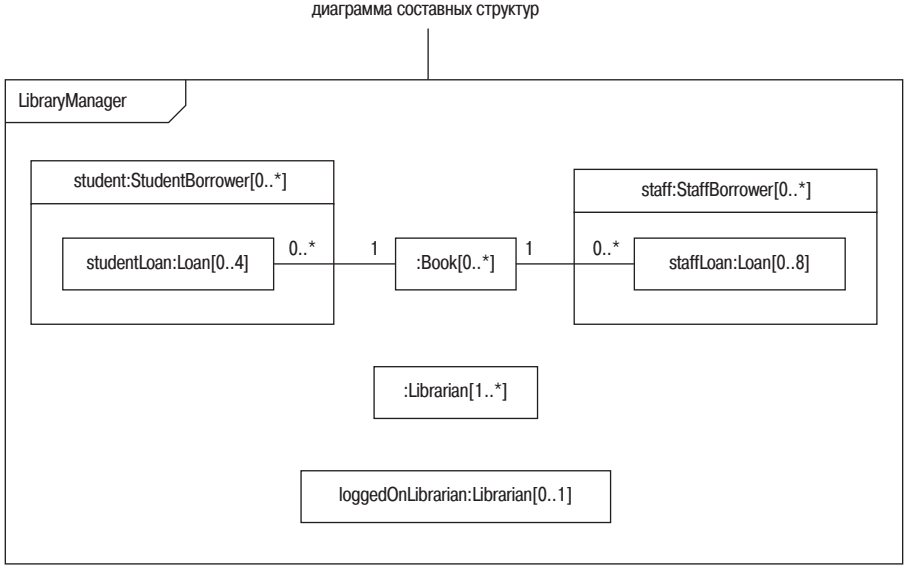


Рис. 18.22. *Диаграмма составных структур класса LibraryManager*

18.13. Что мы узнали

В этой главе было показано, как выявленные при анализе отношения преобразуются в отношения уровня проектирования, которые уже можно реализовывать. Мы узнали следующее.

- Уточнение аналитических отношений до отношений уровня проектирования включает:
  - уточнение ассоциаций до агрегации или композитной агрегации в соответствующих случаях;

- реализацию классов-ассоциаций;
- реализацию ассоциаций один-ко-многим;
- реализацию ассоциаций многие-к-одному;
- реализацию ассоциаций многие-ко-многим;
- реализацию двунаправленных ассоциаций;
- введение возможности навигации;
- введение кратности на обоих концах ассоциации;
- введение имени роли на обоих концах ассоциации или, по крайней мере, на целевом конце;
- использование структурированных классификаторов.

- Агрегация и композиция.
  - Это отношения типа целое-часть, где объекты одного класса играют роль целого, или агрегата, а объекты другого класса играют роль частей:
    - целое использует сервисы частей; части обслуживают запросы целого;
    - целое является доминирующей, контролирующей стороной отношения; часть обычно играет более пассивную роль.
  - Эти отношения транзитивны – если `C` является частью `B` и `B` является частью `A`, тогда `C` является частью `A`.
  - Эти отношения асимметричны:
    - целое никогда – ни прямо, ни косвенно – не может быть частью самого себя;
    - циклы в схеме агрегации недопустимы.
  - Существует два типа отношения агрегации:
    - агрегация;
    - композитная агрегация – обычно ее называют просто композицией.
- Агрегация.
  - Семантика агрегации:
    - агрегат может зависеть от частей, а иногда может существовать независимо от них;
    - части могут существовать независимо от агрегата;
    - агрегат является несколько неполным в случае отсутствия некоторых частей;
    - несколько агрегатов могут владеть частями совместно;
    - возможны иерархии и сети агрегации;
    - целое всегда знает о существовании частей, но если отношение однонаправленное, от целого к части, части не знают о целом.

    - Примером агрегации может быть компьютер и его периферийное оборудование:
        - компьютер слабо связан со своим периферийным оборудованием;
        - периферийное оборудование может появляться и исчезать;
        - периферийное оборудование может использоваться совместно несколькими компьютерами;
        - периферийное оборудование не «принадлежит» ни одному конкретному компьютеру.
- Композиция.
    - Это строгая форма агрегации:
        - части одновременно принадлежат только одному композиту;
        - композит обладает исключительной ответственностью за управление всеми своими частями – ответственностью за их создание и уничтожение;
        - композит также может высвобождать части, если ответственность за них берет на себя другой объект;
        - если композит уничтожается, он должен уничтожить все свои части или передать ответственность за них какому-то другому объекту;
        - каждая часть принадлежит только одному композиту, поэтому возможны только иерархии композиции – сетей композиции не существует.
    - Примером композиции может быть дерево и его листья:
        - листья принадлежат только одному дереву;
        - листья не могут принадлежать нескольким деревьям;
        - когда умирает дерево, умирают и его листья.
    - Часть в композите эквивалентна атрибуту:
        - явная композиция применяется, когда части важны и представляют интерес;
        - атрибуты используется, когда части не важны и не представляют интереса.
- Уточнение аналитических ассоциаций.
    - Аналитические ассоциации должны быть уточнены в одно из отношений агрегации везде, где это возможно. Если при этом в схеме агрегации образуется цикл, должна использоваться ассоциация или зависимость.
    - Процедура уточнения ассоциаций до отношений агрегации:
        - вводятся кратности и имена ролей;

    - принимается решение о том, какая сторона отношения является целым, а какая – частью:
    - рассматривается кратность со стороны целого:
        - если она равна 1, вероятно, можно использовать композицию – если ассоциация имеет семантику композиции, должна применяться композиция;
        - если она не равна 1, *должна* использоваться агрегация;
    - вводится возможность навигации от целого к части.
- Типы ассоциаций.
    - Ассоциация один-к-одному – почти всегда становится композицией. Однако она может быть заменена на атрибут или два класса, которые она связывает, могут быть объединены в один.
    - Ассоциация многие-к-одному:
        - используется агрегация; композиция не может использоваться, поскольку кратность целого – «многие»;
        - проводится проверка на наличие циклов в схеме агрегации.
    - Ассоциация один-ко-многим:
        - здесь на стороне части находится коллекция объектов;
        - используется встроенный массив (большинство ОО языков программирования поддерживают массивы напрямую); обычно они обладают плохой гибкостью, но довольно быстрые;
        - используется класс-коллекция; они характеризуются большей гибкостью, чем встроенные массивы, и поиск по коллекции осуществляется быстрее (в других случаях они медленнее массивов).
    - Коллекции.
        - Это специализированные классы, экземпляры которых могут управлять коллекциями других объектов.
        - Все классы-коллекции имеют операции для:
            - добавления объектов в коллекцию;
            - удаления объектов из коллекции;
            - получения ссылки на объект коллекции;
            - обхода коллекции – прохода по коллекции от первого объекта до последнего.
    - Моделирование с использованием коллекций – существует четыре варианта:
        - класс-коллекция моделируется явно;
        - путем введения свойства в отношение, например {Vector}, инструменту моделирования сообщается, какую коллекцию использовать;

    - путем введения свойства в отношение программисту сообщается требуемая семантика коллекции:
        - {ordered} – элементы коллекции располагаются в строгом порядке;
        - {unordered} – элементы коллекции располагаются не в строгом порядке;
        - {unique} – все элементы коллекции уникальны;
        - {nonunique} – в коллекции допускается дублирование элементов.
    - задача по уточнению отношений один-ко-многим в классы-коллекции передается программистам;
    - нельзя «переусердствовать» в моделировании – выбор конкретного класса-коллекции часто является тактическим решением, которое может быть принято программистами во время реализации.
- Типы коллекций:
    - Коллекции OCL:
        - Bag – {unordered, nonunique};
        - Set – {unordered, unique};
        - OrderedSet – {ordered, unique};
        - Sequence – {ordered, nonunique}.
    - Карта:
        - еще называют словарем;
        - очень быстрый поиск соответствующего значения по заданному ключу;
        - ведет себя как база данных с таблицей, состоящей из двух столбцов – ключа и значения;
        - ключи должны быть уникальными.
- Конкретизированные отношения.
    - Некоторые отношения являются исключительно артефактами анализа и должны быть подготовлены к реализации посредством процесса конкретизации.
    - Ассоциации многие-ко-многим:
        - это отношение конкретизируется в класс;
        - принимается решение, какая часть является целым, и используется агрегация, композиция или ассоциация.
    - Двунаправленные ассоциации:
        - заменяются однонаправленной агрегацией или композицией от целого к части и однонаправленной ассоциацией или зависимостью от части к целому.

- Классы-ассоциации:
  - принимается решение, какая сторона является целым, а какая – частью;
  - заменяется классом (обычно его имя аналогично имени класса-ассоциации);
  - добавляется примечание с ограничением, указывающим на то, что объекты на каждом конце конкретизированного отношения должны образовывать уникальную пару.

- Изучение композиции с использованием структурированных классов.
  - Структурированный классификатор – классификатор (например, класс), имеющий внутреннюю структуру.
    - моделируется как части, объединенные с помощью соединителей:
      - часть – *роль*, которую могут исполнять один или более экземпляров классификатора в контексте структурированного классификатора;
        - имя – имя части;
        - тип – тип объектов, которые могут играть роль;
        - кратность – число объектов, которые одновременно могут играть ту или иную роль;
      - соединитель – отношение между частями в контексте структурированного классификатора.
    - Внутренняя структура может быть отображена на диаграммах классов или на диаграмме составных структур.
  - Структурированный класс:
    - класс, имеющий внутреннюю структуру;
    - имеет дополнительное ограничение относительно того, что ему принадлежат (он имеет неявное отношение включения) все его части, соединители и порты.

19

Интерфейсы и компоненты

19.1. План главы

Данная глава посвящена двум основным понятиям – интерфейсам и компонентам. Эти темы обсуждаются вместе, потому что, как вы увидите, они тесно взаимосвязаны. Кроме того, в разделе 19.10 будет показано, как использование интерфейсов в сочетании со специальным типом компонентов, которые называют подсистемами, обеспечивает возможность создания гибких архитектур систем.

19.2. Деятельность UP: Проектирование подсистемы

Эта глава в основном посвящена деятельности UP Проектирование подсистемы (Design a subsystem). Подсистема – это всего лишь особый тип компонентов, поэтому здесь мы обсуждаем компоненты и компонентно-ориентированную разработку вместе с подсистемами. Как будет показано позже, рассматриваемые вопросы оказывают влияние на другие проектные деятельности UP.

Деятельность UP Проектирование подсистемы представлена на рис. 19.2. Оригинальный рисунок изменен в соответствии с UML 2, в котором подсистемы являются типами компонентов, а не помеченными стереотипами пакетами. Измененные артефакты на рисунке затушеваны.

Деятельность Проектирование подсистемы заключается в разделении системы на максимально независимые друг от друга части. Эти части и называют подсистемами. Посредниками во взаимодействии подсистем выступают интерфейсы. Цель проектирования подсистем – минимизировать связанность в системе, разработав соответствующие интерфейсы, и гарантировать во всех подсистемах правильную реализацию поведения, определенного интерфейсами.

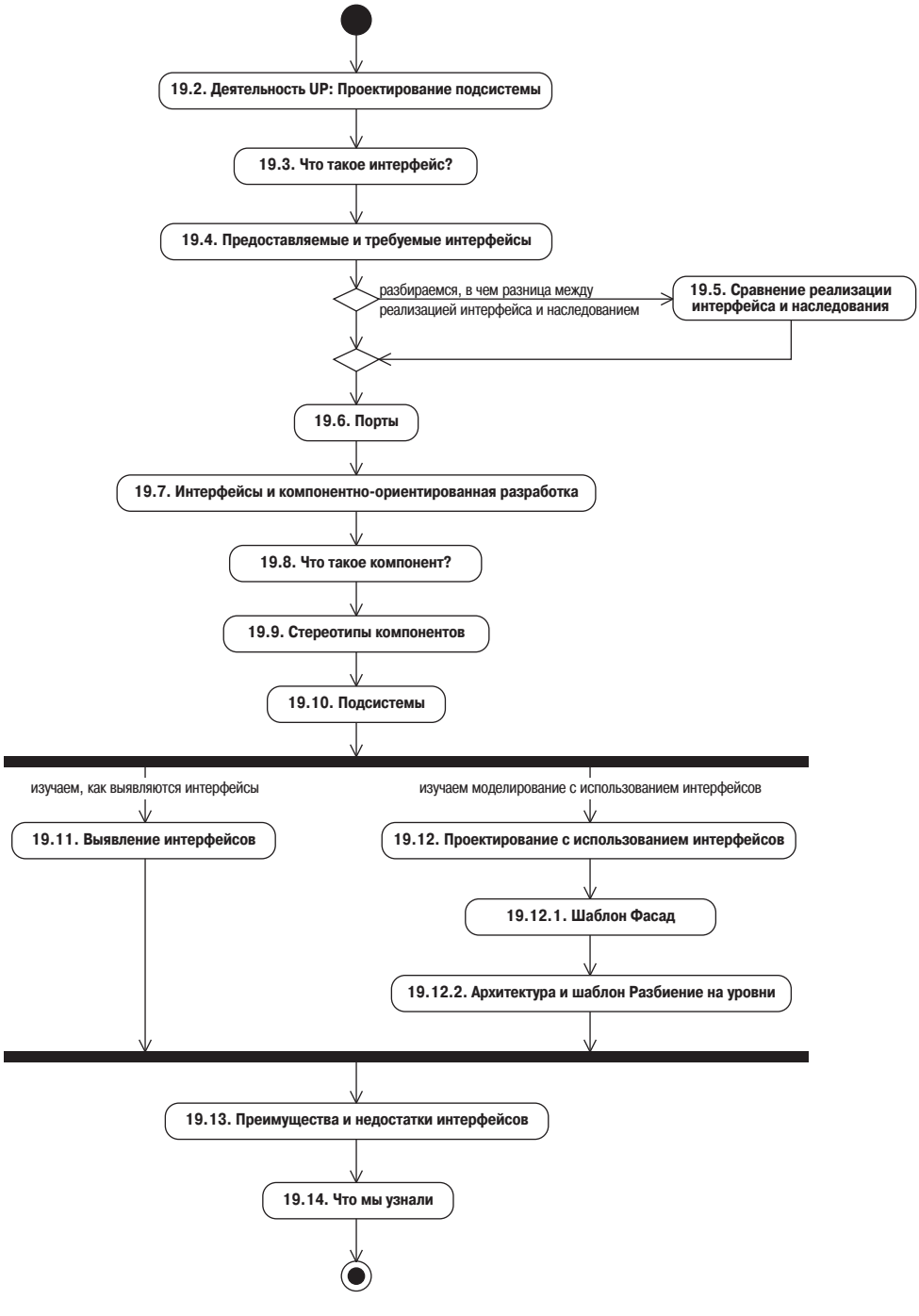


Рис. 19.1. *План главы*

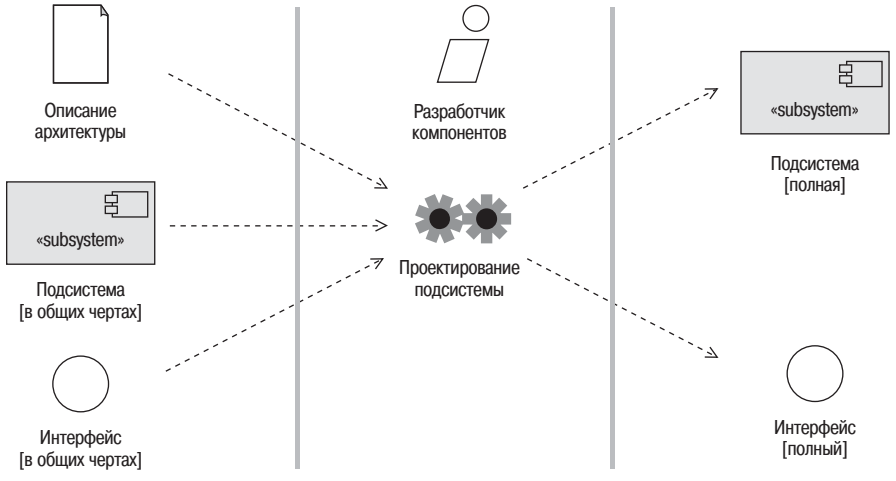


Рис. 19.2. *Деятельность UP Проектирование подсистемы. Адаптировано с рис. 9.44 [Jacobson 1] с разрешения издательства Addison-Wesley*

19.3. Что такое интерфейс?

Интерфейс определяет именованный набор открытых свойств.

> Интерфейс определяет именованный набор открытых свойств.

Главная идея, лежащая в основе интерфейсов, – разделение *описания* функциональности (интерфейс) от ее *реализации* классификатором, таким как класс или подсистема. Создать экземпляр интерфейса невозможно. Он просто объявляет контракт, который может быть реализован классификаторами. Все, что реализует интерфейс, принимает и соглашается следовать определяемому интерфейсом контракту.

> Интерфейсы разделяют описание функциональности от ее реализации.

Возможности, которые могут быть описаны в интерфейсах, перечислены в табл. 19.1. В ней также показаны обязанности реализующих их классификаторов по отношению к интерфейсу.

В интерфейсах также должны присутствовать описания семантики их возможностей (обычно в виде текста или псевдокода) как руководства для тех, кто будет их реализовывать.

Таблица 19.1

Интерфейс описывает	Реализующий классификатор
Операция	Должен иметь операцию с аналогичной сигнатурой и семантикой.
Атрибут	Должен иметь открытые операции для задания и получения значения атрибута – не требуется, чтобы у реализующего классификатора на самом деле был атрибут, определенный интерфейсом, но его поведение должно быть таким, как будто он существует.
Ассоциация	Должен иметь ассоциацию с целевым классификатором – если интерфейс описывает ассоциацию с другим интерфейсом, классификатор, реализующий эти интерфейсы, должен реализовывать ассоциацию между ними.
Ограничение	Должен поддерживать ограничение.
Стереотип	Имеет стереотип.
Помеченное значение	Имеет помеченное значение.
Протокол (например, определенный конечным автоматом протокола – см. раздел 21.2.1)	Должен реализовывать протокол.

Атрибуты и операции в интерфейсе должны быть полностью описаны и включать следующее:

- полную сигнатуру операции (имя, типы всех параметров и возвращаемый тип);
- семантику операции – она может быть записана в виде текста или псевдокода;
- имена и типы атрибутов;
- все стереотипы, ограничения или помеченные значения операции или атрибута.

Важно помнить, что интерфейс определяет только *описание* своих возможностей и что он *никогда* не заключает в себе какой-либо конкретной реализации.

> Интерфейсы определяют контракт.

Интерфейсы могут иметь огромное влияние на способ проектирования. До сих пор проектирование заключалось в соединении определенных классов. Этот процесс можно было бы назвать «проектирование реализации». Однако более гибким подходом является «проектирование контракта», когда классы соединяются с интерфейсом, после чего этот интерфейс может быть реализован любым количеством классов и дру-

гих классификаторов. Стандартные библиотеки Java являются свидетельством гибкости и мощи такого подхода. Часто говорят, что интерфейс определяет *сервис*, предлагаемый классом, подсистемой или компонентом. Современные программные архитектуры часто основываются на сервисах.

> Если язык реализации не поддерживает интерфейсы непосредственно, используйте абстрактные классы.

Язык Java первым из широко используемых языков программирования ввел специальную языковую конструкцию для интерфейсов. Однако интерфейсы вполне могут использоваться при программировании на языках, не имеющих такой специальной конструкции. Например, в C++ просто описывается полностью абстрактный класс, т. е. абстрактный класс, все операции которого абстрактны.

19.4. Предоставляемые и требуемые интерфейсы

Реализуемый классификатором набор интерфейсов называют *предоставляемыми* (*provided*) интерфейсами.

> Реализуемые классификаторами интерфейсы являются предоставляемыми интерфейсами.

Интерфейсы, необходимые для работы классификатора, называют *требуемыми* (*required*) интерфейсами.

> Необходимые классификатору интерфейсы – это требуемые интерфейсы.

На рис. 19.3 показаны два варианта UML-синтаксиса для предоставляемых интерфейсов: нотация в стиле «класса» (в которой можно по-

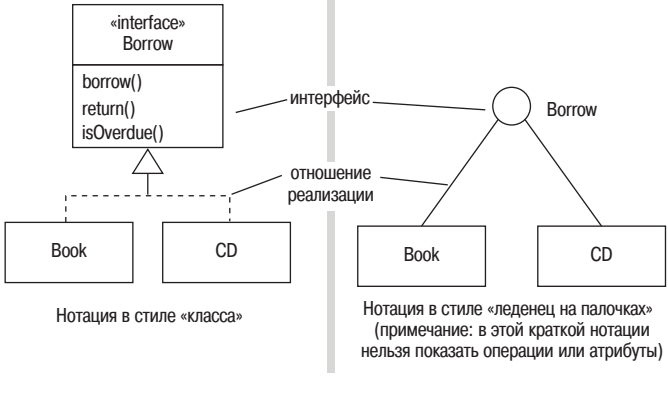


Рис. 19.3. *Два варианта синтаксиса для предоставляемых интерфейсов*

казывать атрибуты и операции) и краткая нотация в стиле «леденец на палочках» (в которой можно *не* показывать атрибуты и операции).

Пример на рис. 19.3 взят из простой системы управления библиотекой, в которой класс обслуживает коллекцию объектов Book и коллекцию объектов CD. Каждый объект Book и CD реализуют интерфейс Borrow, определяющий протокол для позиций, которые могут быть выданы на время. Хотя семантически Book и CD совершенно разные, интерфейс Borrow позволяет Library рассматривать их одинаково, по крайней мере в том, что касается протокола выдачи книг во временное пользование.

Обычно имена интерфейсов и классов совпадают; они записываются в стиле UpperCamelCase. Однако в Visual Basic и C# действует общепринятый стандарт – имя каждого интерфейса начинается с большой I, например IBorrow. Другой стиль присваивания имен – добавление в конце имени суффикса «able», например Borrowable или IBorrowable.

Отношение реализации – это отношение между описанием (в данном случае интерфейсом) и тем, что реализует это описание (в данном случае классами Book и CD). Позже мы увидим, что не только классы могут реализовывать интерфейс. Это также могут делать и другие классификаторы, такие как пакеты и компоненты.

Обратите внимание, что в нотации в стиле класса отношение реализации представляется в виде пунктирной линии с незаштрихованной стрелкой, тогда как в нотации «леденец на палочках» это сплошная линия без стрелки. Главная причина в том, что нотация «леденец на палочках» должна оставаться максимально краткой. Нотацию класса используют, когда хотят показать возможности интерфейса, а нотацию «леденец на палочках» – когда этого делать не надо. Но они обе представляют одно и то же.

На рис. 19.4 показан класс Library, которому требуется интерфейс Borrow. Требуемый интерфейс можно показать как зависимость, направ-

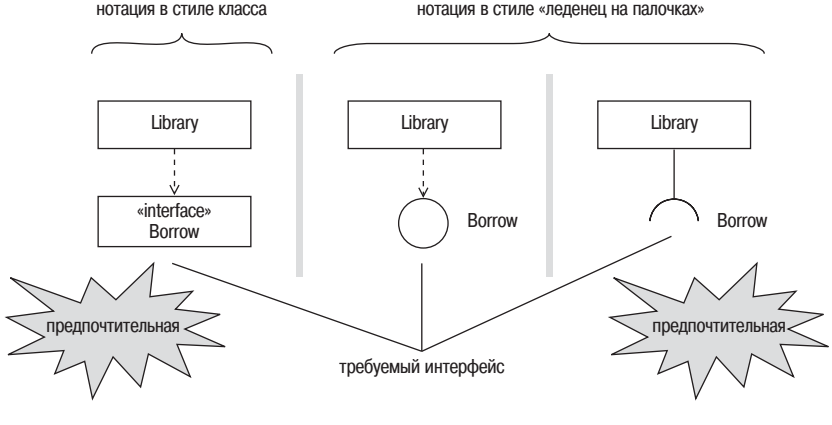


***Рис. 19.4.** Варианты синтаксиса требуемого интерфейса*

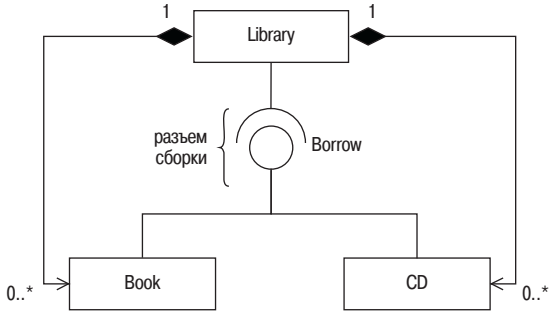


Рис. 19.5. *Система Library в сборке*

ленную к интерфейсу (как в нотации класса, так и в нотации «леденец на палочках»), или как гнездо под требуемый интерфейс. В любом случае эта нотация показывает, что Library понимает и требует конкретный протокол, определенный интерфейсом Borrow. Все элементы, реализующие этот интерфейс, потенциально могут быть подключены к Library, и Library поймет, что они могут быть выданы на время и возвращены.

На рис. 19.5 система Library показана в сборке. Разъем сборки указывает, что и Book, и CD предоставляют необходимый Library набор сервисов (описанных классом Borrow).

Другой хороший пример интерфейсов с множеством реализаций можно найти в классах-коллекциях в стандартных библиотеках Java (рис. 19.6). Хотя там описано всего восемь интерфейсов, Java предоставляет множество разных реализаций, каждая из которых обладает различными характеристиками. При проектировании интерфейса Java-проектировщик может перенести фактическую реализацию интерфейса на период реализации и позволить Java-программисту выбирать реализацию с наиболее подходящими характеристиками.

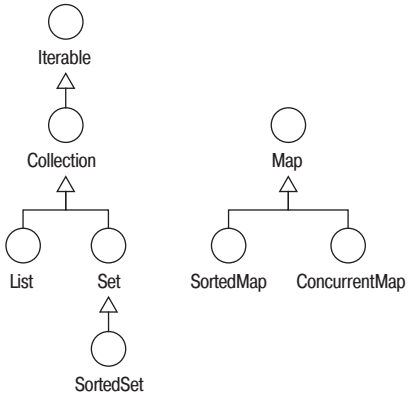


Рис. 19.6. *Интерфейсы с множеством реализаций в классах-коллекциях Java*

19.5. Сравнение реализации интерфейса и наследования

> Реализация интерфейса – «реализует определяемый им контракт».

Теперь стоит обсудить, в чем разница между реализацией интерфейса и наследованием. Семантика реализации интерфейса – «реализует определенный контракт», а семантика наследования – «является». Принцип замещаемости применим как к наследованию, так и к реализации интерфейса. Таким образом, оба типа отношений могут формировать полиморфизм.

> Семантика наследования – «является».

Чтобы проиллюстрировать разницу между реализацией интерфейса и наследованием, мы предлагаем альтернативное решение для системы управления библиотекой на основании наследования (рис. 19.7). Оно кажется вполне приемлемым и в некотором отношении даже более простым, но здесь есть некоторые проблемы.

Прежде всего обсудим, почему эта основанная на наследовании модель не совсем подходит. Здесь очень четко определяется, что объекты Book и CD имеют тип BorrowableItem. Но разве этой способности Book и CD быть выданными на время достаточно для описания их типа? Наверное, ее можно было бы рассматривать как один из аспектов их поведения, который оказался бы общим в контексте библиотечной системы. Семантически правильнее рассматривать BorrowableItem как отдельную *роль*, которую Book и CD играют в Library, а не как общий надтип.

Чтобы конкретизировать недостаток модели, изображенной на рис. 19.7, добавим в систему Library класс Journal (журнал). Journal – это периодическое издание, такое как «Nature» (Природа), не выдаваемое

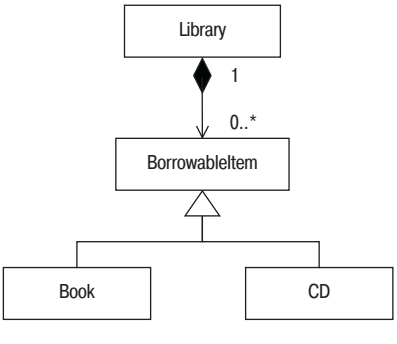


Рис. 19.7. *Модель, основанная на наследовании, имеет недостаток*

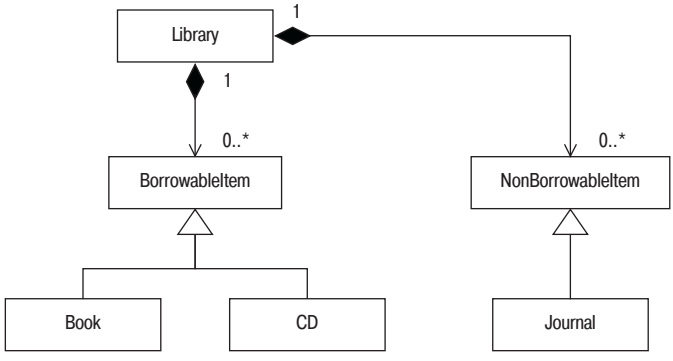


Рис. 19.8. *Модель Library с двумя списками объектов*

на время. В результате получается основанная на наследовании модель, представленная на рис. 19.8.

Обратите внимание, что теперь Library должен обслуживать два списка объектов – те, которые могут, и те, которые не могут выдаваться на время. Такое решение работоспособно, но не очень изящно, поскольку в нем смешиваются два очень разных понятия системы Library:

- хранящиеся объекты;
- объекты, выдаваемые на время.

Эту модель можно несколько усовершенствовать, введя дополнительный уровень иерархии наследования, как показано на рис. 19.9. Использование класса LibraryItem (библиотечная позиция) избавляет от одного из отношений композиции. Такое решение в принципе под-

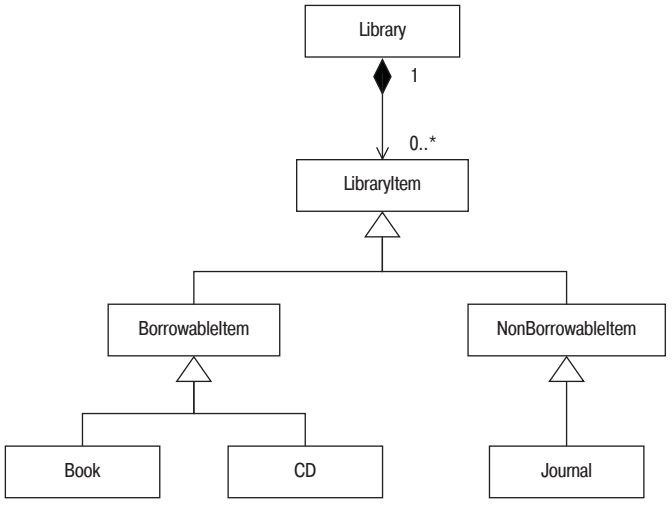


Рис. 19.9. *Усовершенствованная модель Library*

ходит, поскольку в нем используется только наследование. Протокол «выдаваемый на время» вынесен в отдельный уровень иерархии наследования. Это обычное решение проблемы такого рода.

Модель, использующая и интерфейсы, и наследование, обеспечивает более элегантное решение (рис. 19.10).

Основанное на интерфейсах решение имеет следующие преимущества:

- каждая позиция в Library является LibraryItem;
- понятие «возможность выдачи на время» вынесена в отдельный интерфейс, Borrowable, который может применяться к классам LibraryItem в случае необходимости;
- сокращается число классов – пять классов и один интерфейс в противоположность семи классам в другом решении;
- меньше отношений композиции – одно в противоположность двум в другом решении;
- более простая, состоящая всего из двух уровней, иерархия наследования;
- меньше отношений наследования – три в противоположность пяти в другом решении.

В общем, основанное на интерфейсе решение проще и обладает лучшей семантикой. Такие характеристики, как catalogNumber (номер каталога), которые есть у всех LibraryItem, были вынесены в базовый класс, чтобы обеспечить возможность их наследования. А протокол «возможность выдачи на время» определен отдельно в интерфейсе Borrowable.

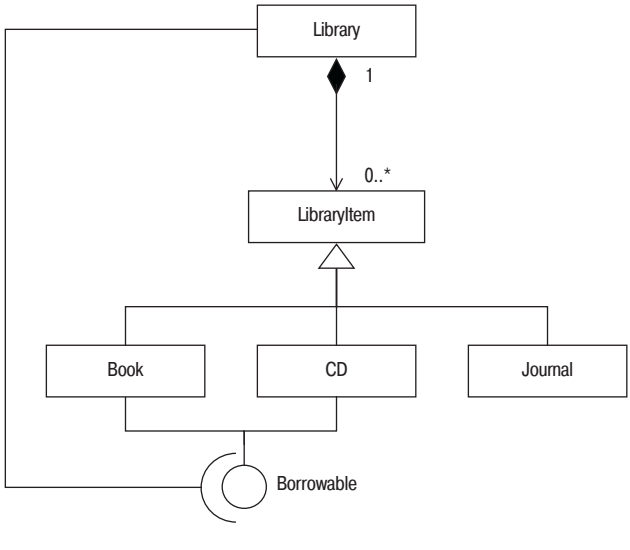


Рис. 19.10. *Модель Library, использующая и интерфейсы, и наследование*

Чтобы проиллюстрировать гибкость интерфейсов, давайте пойдем в этом примере на шаг вперед. Предположим, что необходимо экспортировать данные классов Book и Journal (но не CD) в XML-файлы. Прикладные драйверы в этом случае должны обеспечивать возможность обмена информацией с другими библиотеками и представления каталога печатных материалов в Веб. Было спроектировано следующее решение:

- для осуществления экспорта в XML введен класс XMLExporter;
- введен интерфейс XMLExportable, определяющий протокол для работы с XMLExporter, который должен быть у каждой экспортируемой позиции.

Имеются следующие нефункциональные требования:

- языком реализации должен быть Java;
- для обработки XML должна использоваться библиотека JDOM (JDOM – простая, но мощная библиотека Java для работы с XML-документами; см. *www.jdom.org*).

Протокол XMLExportable – это всего лишь одна операция getElement(), возвращающая представление экспортируемого элемента в виде класса Element, описанного в библиотеке JDOM. Класс XMLExporter использует JDOM для записи объектов Element в XML-файл.

Полностью решение представлено на рис. 19.11.

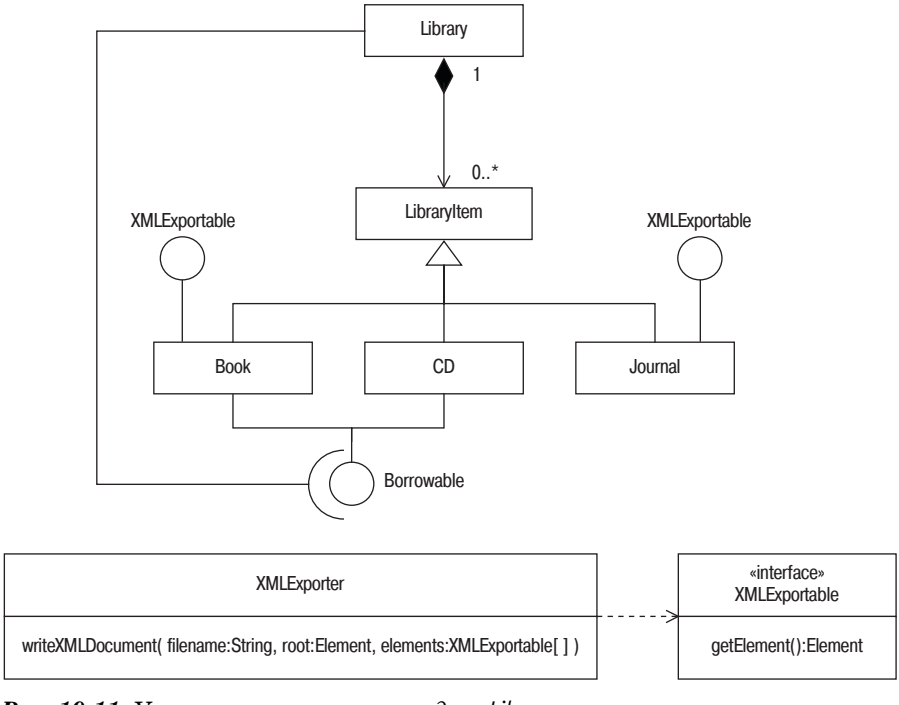


Рис. 19.11. *Усовершенствованная модель Library с возможностью экспорта данных в XML-файлы*

> Интерфейсы используются для описания общих протоколов классов, которые обычно *не* связываются через наследование.

В данном решении удалось разделить вопросы хранения (отношение композиции), выдачи на время (Borrowable) и экспортируемости (XMLExportable). Интерфейсы использовались для определения общих протоколов классов, которые *не* должны быть связаны через наследование.

19.6. Порты

Порт группирует семантически связанный набор предоставляемых и требуемых интерфейсов. Он указывает на конкретную точку взаимодействия классификатора и его окружения.

> Порт группирует семантически связанный набор предоставляемых и требуемых интерфейсов.

Пример на рис. 19.12 иллюстрирует нотацию порта. Здесь показан класс Book, имеющий порт presentation (представление). Этот порт состоит из требуемого интерфейса DisplayMedium (устройство отображения) и предоставляемого интерфейса Display (отображение). Имя порта является необязательным. На рисунке показано два варианта нотации порта: слева – обычная, а справа – более краткий альтернативный вариант. Однако эта альтернатива применима, *только* если у порта о*дин тип* предоставляемого интерфейса (у него по-прежнему может быть нуль или более требуемых интерфейсов). Имя типа указывается после имени порта, как показано на рисунке.

Порты являются очень удобным способом структурирования предоставляемых и требуемых интерфейсов классификатора. Их также можно использовать для упрощения диаграммы. Например, на рис. 19.13 показан класс Viewer (средство просмотра), соединяющийся с портом presentation класса Book. Чтобы обеспечить возможность соединения портов, их предоставляемый и требуемый интерфейсы должны совпадать. Использование портов, очевидно, обеспечивает намного более

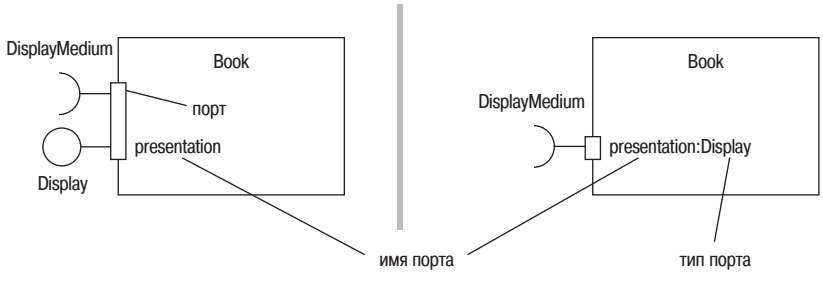


***Рис. 19.12.** Два варианта нотации порта*

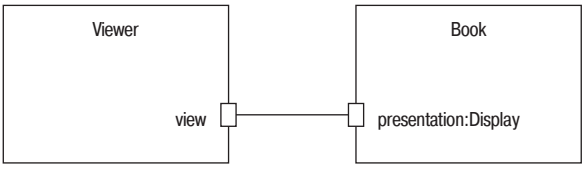


Рис. 19.13. *Соединение портов*

краткое представление, чем отображение всех предоставляемых и требуемых интерфейсов, но может и усложнять чтение диаграмм.

У портов может быть видимость. Когда порт изображается перекрывающим границу классификатора, он является открытым. Это означает, что предоставляемый и требуемый интерфейсы открытые (public). Если прямоугольник порта находится внутри границы классификатора (как показано на рис. 19.14), видимость порта принимает два значения: или защищенный (protected) (по умолчанию), или закрытый (private). Фактическая видимость графически не отображается, но записывается в спецификации порта.

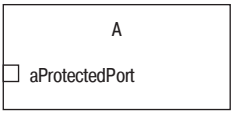


Рис. 19.14. *Защищенный порт*

У порта может быть кратность. Ее указывают в квадратных скобках после имени порта и его типа (например, presentation:Display[1]). Кратность указывает на количество экземпляров порта, которые будут иметь экземпляры классификатора.

19.7. Интерфейсы и компонентно-ориентированная разработка

> Компонентно-ориентированная разработка заключается в построении программного обеспечения из подключаемых частей.

Интерфейсы – ключ к компонентно-ориентированной разработке (component-based development, CBD). Она заключается в построении программного обеспечения из подключаемых частей. Если требуется создать гибкое, ориентированное на компоненты программное обеспечение, для которого по желанию можно подключать новые реализации, при проектировании должны использоваться интерфейсы. Поскольку интерфейсы определяют только контракт, они обеспечивают возможность существования *любого количества* конкретных реализаций, подчиняющихся этому контракту.

В следующих разделах мы расскажем о компонентах, а затем обсудим, как можно сочетать компоненты и интерфейсы в CBD.

19.8. Что такое компонент?

Спецификация UML 2.0 [UML2S] гласит: «Компонент представляет модульную часть системы, которая инкапсулирует ее содержимое, и реализация компонента замещаема в рамках его окружения». Компонент как черный ящик, внешнее поведение которого полностью определяется его предоставляемыми и требуемыми интерфейсами. Поэтому один компонент может быть заменен другим, поддерживающим тот же протокол.

> Компонент – модульная и замещаемая часть системы, инкапсулирующая ее содержимое.

Компоненты могут иметь атрибуты и операции и участвовать в отношениях ассоциации и обобщения. Компоненты – это структурированные классификаторы. У них может быть внутренняя структура, включающая части и соединители. Структурированные классификаторы рассматривались в разделе 18.12.1. Если вы до сих пор не ознакомились с ними, рекомендуем сделать это сейчас, прежде чем двигаться дальше.

Компоненты могут представлять что-то, экземпляр чего может быть создан во время выполнения, например EJB (Enterprise JavaBean). Или ими может быть представлена абсолютно логическая конструкция, такая как подсистема, экземпляры которой создаются только косвенно через создание экземпляров ее частей.

Компонент может быть представлен одним или более артефактами. Артефакт представляет некоторую физическую сущность, например исходный файл. В частности, компонент EJB мог бы быть представлен файлом JAR (Java Archive – Java-архив). Более подробно артефакты обсуждаются в разделе 24.5.

На диаграмме компонентов могут быть показаны компоненты, зависимости между ними и то, как компонентам назначаются классификаторы. Компонент отображается в виде прямоугольника со стереотипом «component» (компонент) и/или пиктограммой компонента в верхнем правом углу, как на рис. 19.15. У компонентов могут быть предоставляемые и требуемые интерфейсы и порты.

Компонент может иметь внутреннюю структуру. Части можно показать вложенными внутрь компонента (рис. 19.16) или находящимися снаружи и соединенными с ним отношением зависимости. Обе формы синтаксически эквивалентны, хотя первая нотация нам кажется более наглядной.

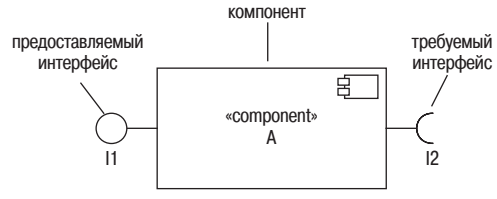


Рис. 19.15. *Нотация компонента*

Если у компонента есть внутренняя структура, как правило, он будет делегировать обязанности, определенные его интерфейсами, своим внутренним частям. На рис. 19.16 компонент A предоставляет интерфейс I1 и требует интерфейс I2. Он инкапсулирует две части типа b и c. Он делегирует поведение, описанное его предоставляемым и требуемым интерфейсами b и c соответственно.

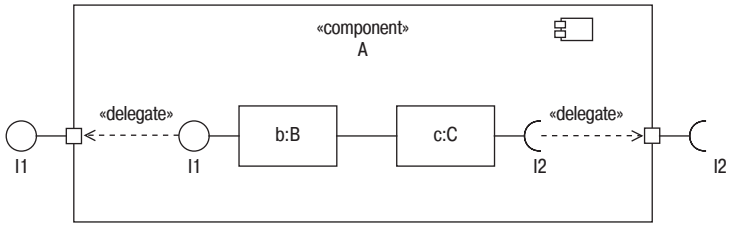


Рис. 19.16. *Внутренняя структура компонента*

> Интерфейсы позволяют гибко объединять компоненты.

Компоненты могут зависеть от других компонентов. Для разъединения компонентов в качестве посредников в зависимости *всегда* используются интерфейсы. Если компоненту требуется интерфейс, представить это можно в виде зависимости между компонентом и интерфейсом либо использовать разъем сборки, как показано на рис. 19.17.

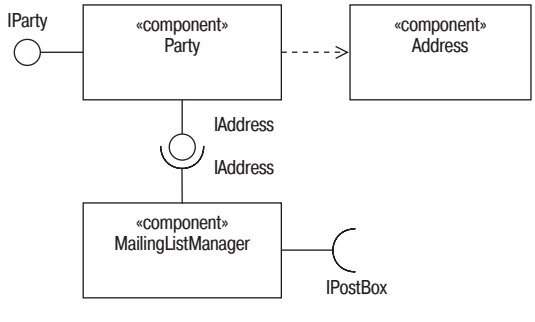


Рис. 19.17. *Разъем сборки между компонентами*

На рис. 19.17 показано следующее:

- Компонент Party предоставляет два интерфейса типа IParty (вечеринка) и IAddress (адрес). Эти интерфейсы отображены в виде кружков.
- Компоненту MailingListManager (менеджер рассылки писем) требуются два интерфейса типа IAddress и IPostBox (почтовый ящик). Они представлены в виде гнезд.
- Между компонентами Party и MailingListManager – разъем сборки. Он показывает, что MailingListManager общается с компонентом Party посредством предоставляемого интерфейса IAddress.
- В этой модели компонент Party играет роль фасада (см. раздел 19.12.1) для разъединения компонента MailingListManager и деталей компонента Address.

Часто компоненты отображаются просто как «черные ящики», к которым прикреплены их предоставляемые и требуемые интерфейсы. Однако компонент может быть представлен и как «белый ящик» (рис. 19.18). Такое полное представление раскрывает внутренние детали компонента. В нем могут быть показаны любые предоставляемые интерфейсы, требуемые интерфейсы, реализации или ассоциированные артефакты.

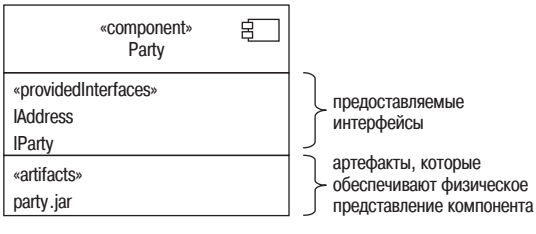


***Рис. 19.18.** Полное представление компонента*

19.9. Стереотипы компонентов

Компоненты – вероятно, наиболее богатые стереотипами элементы UML. Это объясняется тем, что компоненты могут использоваться для представления различных типов сущностей. UML 2 предоставляет небольшой набор стандартных стереотипов компонентов, которые перечислены в табл. 19.2. Один из них, «subsystem», рассматривается более подробно в следующем разделе.

Если при моделировании используются профили UML, они определяют собственные стереотипы компонентов.

Таблица 19.2

Стереотип	Семантика
«buildComponent»	Компонент, определяющий набор сущностей для организационных или системных целей разработки.
«entity»	Компонент постоянной информации, представляющий бизнес-понятие.
«implementation»	Определение компонента, не имеющего собственной спецификации. Он является реализацией отдельного компонента, обозначенного стереотипом «specification», с которым имеет отношение зависимости.
«specification»	Классификатор, определяющий область объектов без описания физической реализации этих объектов. Например, компонент, обозначенный стереотипом «specification», имеет только предоставляемые и требуемые интерфейсы и не имеет реализующих классификаторов.
«process»	Компонент, ориентированный на транзакции.
«service»	Не имеющий состояния функциональный компонент, вычисляющий значение.
«subsystem»	Единица иерархической декомпозиции больших систем.

19.10. Подсистемы

Подсистема – это компонент, действующий как единица декомпозиции большой системы. Подсистемы изображаются как компонент со стереотипом «subsystem».

> Подсистема – это компонент, действующий как единица декомпозиции большой системы.

Подсистема – это логическая конструкция, используемая для декомпозиции большой системы в управляемые части. Экземпляры самих подсистем *не могут* создаваться во время выполнения, но могут создаваться экземпляры их содержимого.

С точки зрения UP подсистемы являются ключевой концепцией структурирования. Деление системы на подсистемы позволяет разложить большую, сложную задачу разработки на много меньших и более управляемых подзадач. Это ключ к успешной разработке системы с использованием UP.

> Интерфейсы используются для сокрытия деталей реализации подсистем.

Интерфейсы идут рука об руку с подсистемами, как показано на рис. 19.19. В данном примере подсистеме GUI известны *только* интер-

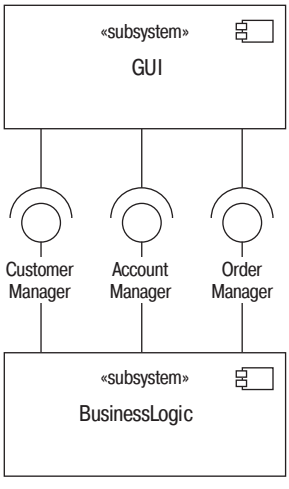


Рис. 19.19. *Подсистемы и интерфейсы*

фейсы CustomerManager, AccountManager и OrderManager. Она ничего не знает о внутренней реализации подсистемы BusinessLogic (бизнес-логика). Это значит, что в принципе подсистему BusinessLogic можно было бы полностью заменить даже несколькими подсистемами, если вместе они будут предоставлять такой же набор интерфейсов. Аналогично можно было бы заменить подсистему GUI другой подсистемой GUI, требующей такой же набор интерфейсов. Такое использование интерфейсов разъединяет подсистемы и обеспечивает гибкость архитектуры.

Интерфейсы объединяют подсистемы, создавая архитектуру системы.

19.11. Выявление интерфейсов

При проектировании системы или ее части следует проверить проектную модель на предмет выявления некоторых интерфейсов. Это довольно просто сделать, выполнив следующее:

- Каждая ассоциация должна быть поставлена под сомнение – для каждой задается вопрос: «Действительно ли данная ассоциация должна быть установлена с конкретным классом объектов или она должна быть более гибкой?» Если решено, что ассоциация должна быть более гибкой, чем она была бы в случае привязки к конкретному классу, необходимо рассмотреть возможность использования интерфейса.
- Каждое сообщение должно быть поставлено под сомнение – для каждого задается вопрос: «Действительно ли данное сообщение должно отправляться объектам только одного класса или оно должно быть более гибким?» Если оно должно быть более универсальным (т. е. если можно найти классы, которые могли бы отправлять такие же

сообщения объектам других классов), необходимо рассмотреть возможность использования интерфейса.

- Выделить группы операций, которые могут повторно использоваться где-либо еще. Например, если многим классам системы необходима возможность работы с некоторым устройством вывода, следует подумать о проектировании интерфейса Print.
- Выделить наборы операций, повторяющиеся в нескольких классах.
- Выделить наборы атрибутов, повторяющиеся в нескольких классах.
- Провести поиск классов, играющих в системе одинаковую роль – роль может указывать на возможный интерфейс.
- Рассмотреть возможности будущего расширения системы. Иногда даже после небольшого предварительного анализа можно спроектировать легко расширяемые в будущем системы. Ключевой вопрос: «Понадобится ли добавлять к системе какие-либо классы в будущем?» Если ответ положительный, необходимо попытаться определить один или более интерфейсов, которые будут описывать протокол добавления этих новых классов.
- Рассмотреть зависимости между компонентами – везде, где это возможно, в них должны быть введены разъемы сборки в качестве посредников.

Как видите, существует много возможностей для использования интерфейсов. Подробности проектирования с применением интерфейсов рассматриваются в следующем разделе.

19.12. Проектирование с использованием интерфейсов

При проектировании весьма полезно, если сущности ведут себя максимально одинаково. Применяя интерфейсы, можно проектировать общие протоколы, которые могли бы реализовываться многими классами или компонентами. Хороший пример этому – система, которую мы разработали, чтобы предоставить общий интерфейс для нескольких унаследованных систем. Проблема состояла в том, что у каждой системы был свой протокол связи. Нам удалось скрыть эту сложность за единственным интерфейсом, состоящим из операций open(…), read(…), write(…) и close().

Приведем другой пример. Рассмотрим систему, моделирующую организацию (например, систему управления человеческими ресурсами). В ней много классов сущностей, имеющих имя и адрес, например Person, OrganizationalUnit, Job. Все эти классы могут играть общую роль addressableUnit (элемент с адресом). Вполне логично, что у всех классов должен быть один и тот же интерфейс для обработки имени и адреса. Следовательно, можно определить интерфейс NameAndAddress (имя и адрес), который могли бы реализовывать все эти классы. В других реше-

ниях этой задачи могло бы использоваться наследование, но решение, основанное на интерфейсе, иногда является более гибким.

Стоит вспомнить, что у классов могут быть рефлексивные ассоциации (на самих себя) и могут существовать внутренние по отношению к классу роли. Эти ассоциации и роли также являются кандидатами в интерфейсы.

Мощь применения интерфейсов состоит в предоставлении возможности подключения элементов в системы. Один из путей сделать систему гибкой и способной изменяться – спроектировать ее так, чтобы обеспечить простое подключение расширений. Интерфейс – ключ к этому. Если есть возможность проектировать системы с использованием интерфейсов, значит, ассоциации и сообщения будут привязаны не к объектам конкретного класса, а к определенному интерфейсу. Это упрощает добавление в систему новых классов, поскольку интерфейсы определяют протоколы, которые должны поддерживаться новыми классами, чтобы из можно было подключить.

Подключаемые алгоритмы – хороший пример программных модулей, которые можно подключать по желанию. Некоторое время назад мы работали над системой, которая проводила большое и сложное вычисление над очень большим набором данных. Пользователи хотели экспериментировать с алгоритмом вычисления в попытках найти оптимальную стратегию. Однако это не было учтено при создании системы. В результате на каждое небольшое изменение алгоритма требовалось несколько человеко-дней, поскольку приходилось менять существующий код и компоновать систему заново. Вместе с одним из проектировщиков системы мы реорганизовали систему и ввели в нее интерфейс для подключаемых алгоритмов. После этого можно было пробовать новые алгоритмы без всяких ограничений. По сути, можно было переключать алгоритмы даже в процессе выполнения системы.

19.12.1. Шаблон Фасад

> Шаблон Фасад скрывает сложную реализацию за простым интерфейсом.

Сокрытие сложных подсистем за хорошо структурированным простым интерфейсом известно как шаблон Фасад (Fasade). Он задокументирован в [Gamma 1]. Эта книга является ценным источником мощных многократно используемых шаблонов, которые могут применяться в проектных моделях во многих разных контекстах. Вот что сказал Гамма (Gamma) о шаблоне Фасад: «Структурирование системы в подсистемы помогает снизить сложность. Общая цель проектирования – свести до минимума общение и зависимости между подсистемами. Один из способов достижения этой цели – введение фасадного объекта, предоставляющего единственный упрощенный интерфейс для наиболее общих возможностей подсистемы».

Шаблон Фасад обеспечивает возможность сокрытия информации и разделения задач – сложные детали внутренней работы подсистемы можно спрятать за простым интерфейсом. Это упрощает систему и позволяет контролировать и управлять связанностью (coupling) подсистем.

Интерфейсы, используемые в качестве фасада, могут применяться для создания «швов» в системе. Это делается следующим образом:

- выявляются связные части системы;
- они упаковываются в «subsystem»;
- определяется интерфейс для этой подсистемы.

19.12.2. Архитектура и шаблон Разбиение на уровни

> Шаблон Разбиение на уровни организовывает подсистемы в семантически связанные уровни.

Коллекция подсистем и интерфейсов уровня проектирования образует высокоуровневую архитектуру системы. Однако, для того чтобы эта архитектура была проста для понимания и обслуживания, коллекция подсистем и интерфейсов по-прежнему должна быть организована логически последовательно. Это можно выполнить с помощью архитектурного шаблона под названием Разбиение на уровни (layering).

Шаблон Разбиение на уровни компонует проектные подсистемы и интерфейсы в уровни. При этом подсистемы каждого уровня семантически связаны. Суть создания устойчивой многоуровневой архитектуры – управление связанностью подсистем благодаря:

- введению новых интерфейсов там, где необходимо;
- такой переорганизации классов в новые подсистемы, которая сокращает количество взаимосвязей между подсистемами.

С зависимостями между уровнями необходимо обращаться очень аккуратно, поскольку они представляют взаимосвязь уровней. В идеале уровни должны быть *максимально разъединенными*, поэтому попытайтесь обеспечить, чтобы:

- зависимости были направлены в одну сторону;
- во всех зависимостях присутствовали интерфейсы-посредники.

Иначе говоря, подсистема определенного уровня по возможности должна требовать интерфейсы от нижележащего уровня и предоставлять интерфейсы более высокому уровню.

Существует много способов создания многоуровневых архитектур. Уровней может быть столько, сколько нужно. Однако в основном системы разделяют на представление, бизнес-логику и сервисные уровни. Как показано на рис. 19.20, также довольно распространено более глубокое разбиение уровня бизнес-логики. В данном случае имеется два уровня – предметная область и сервисы. Уровень предметной области

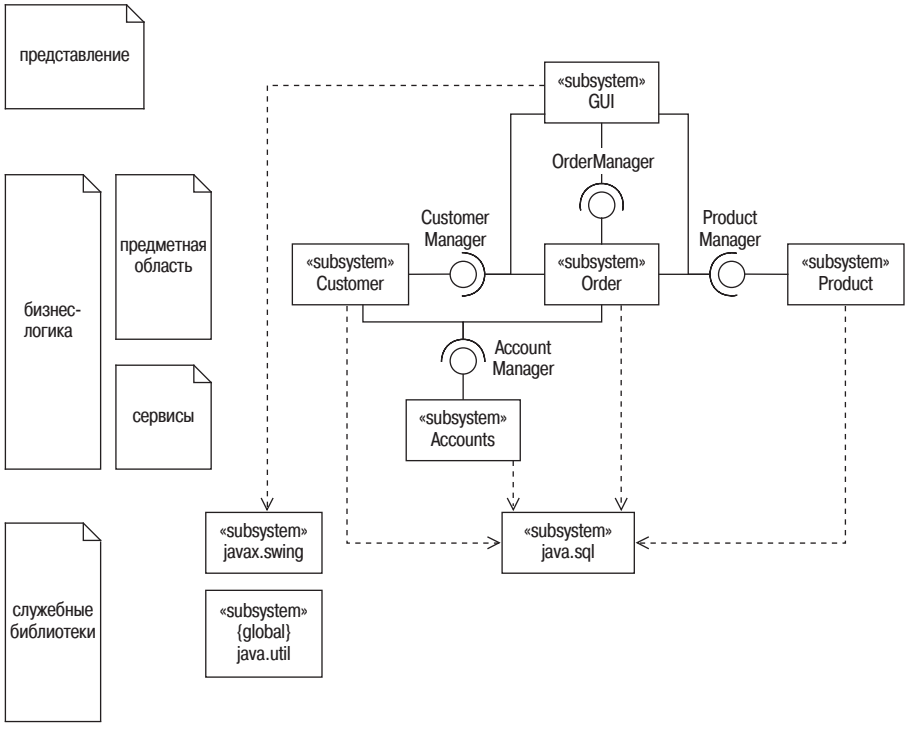


Рис. 19.20. *Трехуровневая архитектура системы*

включает подсистемы, характерные для данного конкретного приложения. В уровень сервиса вошли подсистемы, которые могут повторно использоваться в других приложениях.

Везде, где это возможно, лучше проектировать интерфейс. На рис. 19.20 показано, что все разработанные нами подсистемы соединяются посредством интерфейсов, а пакеты Java просто объединены зависимостями, хотя каждый предоставляет несколько интерфейсов. Причина в том, что если отображение собственных интерфейсов представляет некоторый интерес и пользу, то показывать интерфейсы стандартных библиотек Java не имеет никакого практического смысла. Также обратите внимание, что пакет java.util, содержащий такие универсальные компоненты, как Date, используется повсеместно, поэтому обозначен меткой {global} (глобальный). Эта метка указывает на то, что все открытое содержимое данного пакета является видимым везде. Таким способом показывается, что было решено *не* отображать зависимости к этому пакету, поскольку они не дают никакой полезной информации.

19.13. Преимущества и недостатки интерфейсов

> Проектирование контракта – более гибкий подход, чем проектирование реализации.

При проектировании с использованием классов мы ограничиваемся определенными реализациями. Но когда проектирование ведется с использованием интерфейсов, разрабатываются контракты, которые могут быть реализованы множеством разных реализаций. Проектирование контрактов освобождает модель (и соответственно систему) от зависимости от реализации и, следовательно, увеличивает ее гибкость и расширяемость.

Проектирование с использованием интерфейсов позволяет сократить число зависимостей между классами, подсистемами и компонентами и, следовательно, предоставляет возможность управлять количеством взаимосвязей в модели. Связанность (coupling) на самом деле является злейшим врагом разработчика объектов, поскольку высоковзаимосвязанные системы тяжело понимать, обслуживать и развивать. Соответствующее использование интерфейсов может помочь сократить связанность и разбить модель на связные подсистемы.

> Гибкость может стать причиной сложности, поэтому будьте осторожны.

Однако в использовании интерфейсов есть и недостатки. Как правило, если что-то становится более гибким, оно становится и более сложным. Поэтому при проектировании с использованием интерфейсов необходимо искать компромисс между гибкостью и сложностью. В принципе, сделать интерфейсом можно каждую операцию каждого класса, но понять такую систему было бы просто невозможно! Кроме того, часто жертвой гибкости становится производительность, но обычно ее потери малы в сравнении с увеличением сложности.

При проектировании системы в программном обеспечении стараются представить вполне определенный набор бизнес-семантики. Какая-то часть этой семантики подвижна и меняется довольно быстро, тогда как другая относительно стабильна. Для обработки этих подвижных аспектов необходима гибкость. Но системы можно упростить, ограничившись определенной степенью гибкости для более стабильных частей. В некотором роде это один из секретов хорошего ОО анализа и проектирования: выявление стабильных и нестабильных частей системы и соответствующее их моделирование.

Откровенно говоря, правильное моделирование системы важнее, чем ее гибкость. Всегда основное внимание необходимо уделять прежде всего правильному моделированию ключевой бизнес-семантики и только *потом* думать о гибкости. Запомните правило KISS – keep interfaces sweet and simple (интерфейсы должны быть удобными и простыми)!

19.14. Что мы узнали

Интерфейсы обеспечивают возможность проектировать программное обеспечение как контракт, а не как конкретную реализацию. Мы узнали следующее:

- Деятельность UP Проектирование подсистемы заключается в разделении системы на подсистемы – по возможности максимально независимые части.
  - Посредниками во взаимодействии систем выступают интерфейсы.
- Интерфейс определяет именованный набор открытых свойств.
  - Интерфейсы отделяют описание функциональности от ее реализации.
  - Интерфейсы могут быть прикреплены к классам, подсистемам, компонентам и любому другому классификатору и определяют предлагаемые ими сервисы.
  - Если классификатор в подсистеме реализует открытый интерфейс, подсистема или компонент также реализует открытый интерфейс.
  - Все, что реализует интерфейс, соглашается придерживаться контракта, определенного набором операций, описанных в интерфейсе.
- Семантика интерфейса – реализующий интерфейс классификатор имеет следующие обязанности по отношению к каждой из возможностей:
  - операция – должен иметь операцию с аналогичной сигнатурой и семантикой;
  - атрибут – должен иметь открытые операции для задания и получения значения атрибута:
    - классификатор, реализующий интерфейс, *не* обязан иметь атрибут, но он должен вести себя так, как будто атрибут у него есть;
  - ассоциация – должен иметь ассоциацию с целевым классификатором:
    - если интерфейс описывает ассоциацию с другим интерфейсом, ассоциация между этими интерфейсами должна сохраняться и у реализующего их классификатора;
  - ограничение – должен поддерживать ограничение;
  - стереотип – имеет стереотип;
  - помеченное значение – имеет помеченное значение;
  - протокол – должен реализовывать протокол.

- Проектирование реализации:
  - соединяются конкретные классы;
  - чтобы сохранить простоту (но не гибкость), необходимо проектировать реализацию.
- Проектирование контракта:
  - класс соединяется с интерфейсом, у которого может быть множество возможных реализаций;
  - чтобы обеспечить гибкость (что, вероятно, повысит сложность), необходимо проектировать контракт.
- Предоставляемый интерфейс – интерфейс, предоставляемый классификатором:
  - классификатор реализует интерфейс;
  - если в модели необходимо показать операции, используется нотация в стиле «класса»;
  - если интерфейс отображается без операций, используется нотация в стиле «леденец на палочках».
- Требуемый интерфейс – интерфейс, требуемый классификатором:
  - классификатору необходим другой классификатор, реализующий этот интерфейс;
  - отображается зависимость на интерфейс, представленный в виде класса либо с помощью нотации «леденец на палочках», или используется разъем сборки.
- Разъем сборки – объединяет предоставляемый и требуемый интерфейсы.
- Сравнение реализации интерфейса с наследованием:
  - реализация интерфейса – «реализует определенный контракт»;
  - наследование – «является»;
  - и наследование, и реализация интерфейса формируют полиморфизм;
  - интерфейсы используются для описания общих протоколов классов, которые обычно не связываются через наследование.
- Порт – группирует семантически связный набор предоставляемых и требуемых интерфейсов:
  - может иметь имя, тип и видимость.
- Компонент – модульная часть системы, инкапсулирующая ее содержимое, реализация компонента замещаема в рамках его окружения:
  - может иметь атрибуты и операции;
  - может участвовать в отношениях;
  - может иметь внутреннюю структуру;

  - его внешнее поведение полностью определяется его предоставляемыми и требуемыми интерфейсами;
  - компоненты представляют один или более артефактов.
- Компонентно-ориентированная разработка (CBD) занимается построением программного обеспечения из подключаемых частей:
  - чтобы сделать компоненты «подключаемыми», применяются интерфейсы;
  - проектирование интерфейса позволяет использовать много разных реализаций множеством различных компонентов.
- Компоненты могут представлять:
  - физическую сущность (например, EJB-компонент);
  - логическую сущность (например, подсистему).
- Стандартные стереотипы компонентов:
  - «buildComponent» – компонент, определяющий набор сущностей для организационных или системных целей разработки;
  - «entity» – компонент постоянной информации, представляющий бизнес-понятие;
  - «implementation» – компонент, не имеющий собственной спецификации; он является реализацией отдельного компонента, обозначенного стереотипом «specification», с которым имеет отношение зависимости;
  - «specification» – классификатор, который определяет набор объектов без описания их физической реализации – например, компонент, обозначенный стереотипом «specification», имеет только предоставляемые и требуемые интерфейсы, но не имеет реализующих классификаторов;
  - «process» – ориентированный на транзакции компонент;
  - «service» – не имеющий состояния функциональный компонент, вычисляющий значение;
  - «subsystem» – элемент иерархической декомпозиции больших систем.
- Подсистема – компонент, который играет роль элемента декомпозиции большой системы:
  - компонент, обозначенный стереотипом «subsystem»;
  - логическая конструкция, используемая для разложения большой системы на управляемые части;
  - во время выполнения экземпляр подсистемы *не может* быть создан, но экземпляры ее содержимого создать можно;
  - разложение системы на подсистемы – ключ к успешной разработке системы с использованием UP.

- Подсистемы используются для:
  - разделения задач проектирования;
  - представления мало детализированных компонентов;
  - создания оболочек для унаследованных систем.
- Интерфейсы используются для сокрытия деталей реализации подсистем:
  - шаблон Фасад скрывает сложную реализацию за простым интерфейсом;
  - шаблон Разбиение на уровни компонует подсистемы в семантически связные уровни:
    - зависимости между уровнями должны быть направлены в одну сторону;
    - во всех зависимостях между уровнями должны присутствовать интерфейсы-посредники;
    - примерами уровней могут быть представление, бизнес-логика и сервисные уровни.
- Выявление интерфейсов:
  - ставим под сомнение ассоциации;
  - ставим под сомнение отправки сообщений;
  - выделяем группы многократно используемых операций;
  - выделяем группы повторяющихся операций;
  - выделяем группы повторяющихся атрибутов;
  - ищем классы, играющие в системе одну и ту же роль;
  - ищем возможности будущего расширения;
  - ищем зависимости между компонентами.

20

Реализация прецедента на этапе проектирования

20.1. План главы

В этой главе рассматривается проектная реализация прецедента. Это процесс уточнения аналитических диаграмм взаимодействий и классов, в результате которого на них будут представлены артефакты проектирования. Проектные классы были подробно рассмотрены в главе 17, а здесь основное внимание обращено на диаграммы взаимодействий. В частности, обсуждается использование диаграмм взаимодействий в проектировании для моделирования центральных механизмов. Последние представляют собой стратегические проектные решения, которые необходимо принять относительно сохранения (persistence) объектов, их распределения (distribution) и т. д. Кроме того, на примере создания диаграмм взаимодействия подсистем мы научимся использование диаграммы взаимодействий для представления высокоуровневых взаимодействий в системе. В этой главе рассказывается о временных диаграммах. Это новый, введенный в UML 2 тип диаграмм, очень полезный для моделирования аппаратных систем реального времени и встроенных систем. И завершается глава реальным, но простым примером реализации прецедента на этапе проектирования.

20.2. Деятельность UP: Проектирование прецедента

Деятельность UP Проектирование прецедента (Design a use case) (рис. 20.2) заключается в выявлении проектных классов, интерфейсов и компонентов, взаимодействие которых обеспечивает поведение, описанное прецедентом (артефакты, измененные по сравнению с оригинальным рисунком, затушеваны). Это процесс реализации прецедента, который

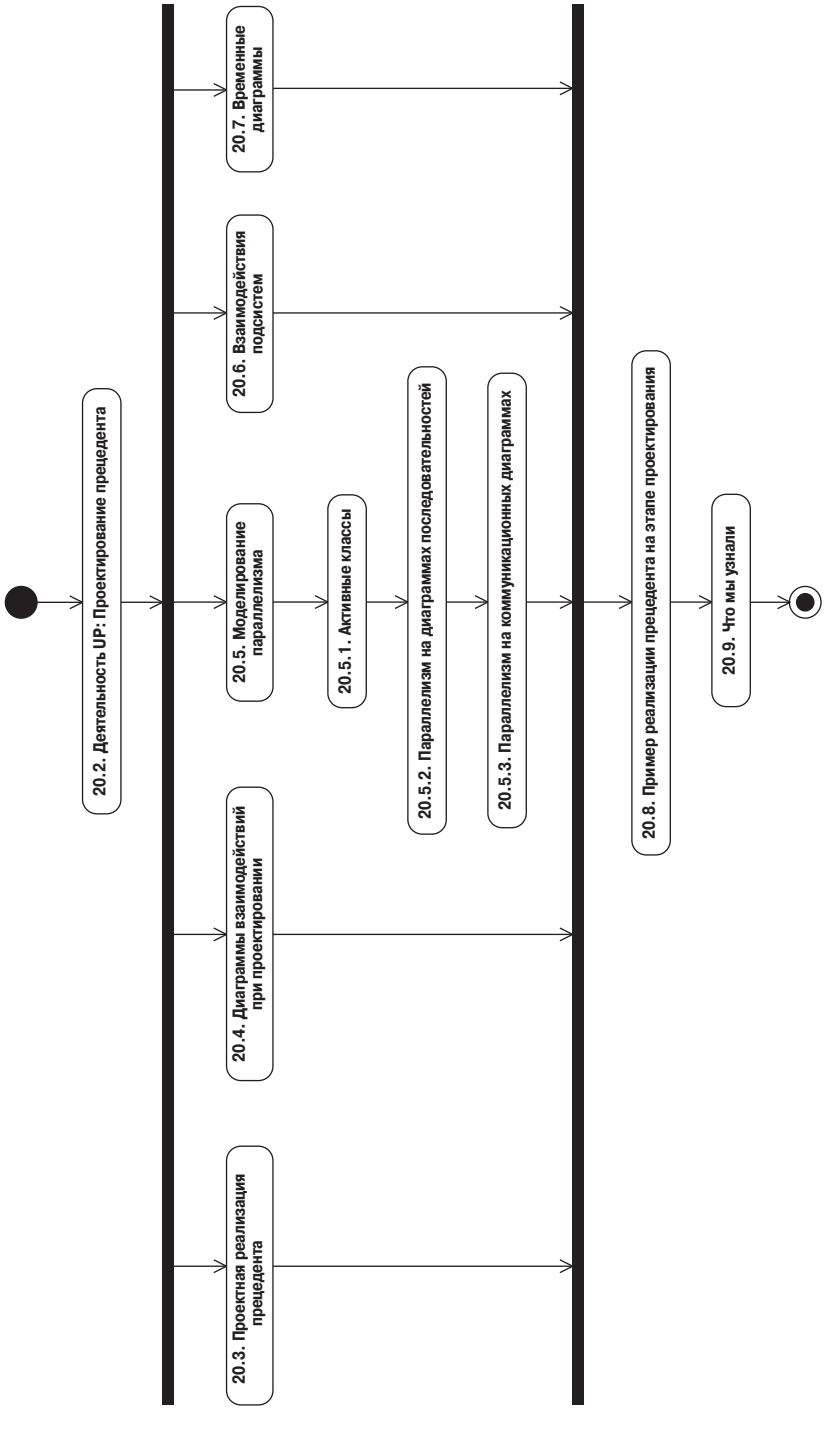


Рис. 20.1. План главы

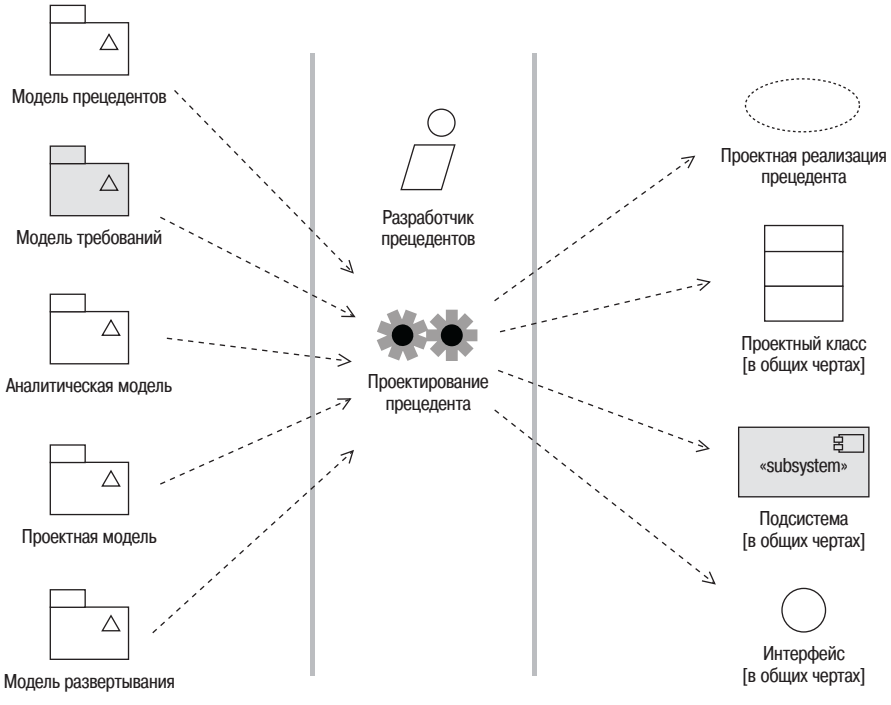


Рис. 20.2. *Деятельность UP Проектирование прецедента. Адаптировано с рис. 9.34 [Jacobson 1] с разрешения издательства Addison-Wesley*

обсуждался в главе 12, но теперь основное внимание сосредоточено на проектировании, что имеет несколько важных следствий:

- При проектировании в реализациях прецедентов участвуют проектные классы, интерфейсы и компоненты, а не классы анализа.
- Процесс создания реализаций прецедентов на этапе проектирования часто выявляет новые нефункциональные требования и новые проектные классы.
- Проектирование реализации прецедента помогает найти то, что Буч называет центральными механизмами [Booch 1]. Это стандартные пути решения конкретных проблем проектирования (например, организация доступа к базе данных), которые остаются *неизменными* в течение всего процесса разработки.

Входными артефактами Проектирования прецедента являются:

- Модель прецедентов – см. часть II «Определение требований».
- Модель требований – см. часть II «Определение требований».
- Аналитическая модель – рассматривается в части III «Анализ».
- Проектная модель – это то, что мы создавали в разделе, посвященном проектированию. UP представляет проектную модель как входной

артефакт Проектирования прецедента, чтобы обозначить итеративную природу этого процесса. По мере выявления в процессе проектирования все большего числа деталей системы происходит уточнение каждого из артефактов.

- Модель развертывания – обсуждается в главе 24. Модель развертывания также представлена как входной артефакт этой деятельности проектирования, чтобы проиллюстрировать, как все артефакты совместно эволюционируют во времени.

Важно понимать, что проектирование – итеративный процесс, а не последовательность шагов. По существу, информация, выявленная в отношении одного артефакта, может повлиять на остальные артефакты. Синхронизация всех артефактов является составной частью проектирования.

20.3. Проектная реализация прецедента

> «Проектная реализация прецедента» – это взаимодействие проектных объектов и проектных классов, реализующих прецедент.

Проектная реализация прецедента – это взаимодействие проектных объектов и проектных классов, реализующих прецедент. Между аналитической и проектной реализациями прецедента установлено отношение «trace». Проектирование реализации прецедента определяет решения уровня реализации и реализует нефункциональные требования. Проектная реализация прецедента состоит из:

- проектных диаграмм взаимодействий;
- диаграмм классов, включающих участвующие в ней проектные классы.

При анализе основное внимание в реализации прецедентов было сосредоточено на том, *что* должна делать система. В проектировании нас интересует, *как* система собирается это делать. Таким образом, теперь нам необходимо определить детали реализации, которые игнорировались на этапе анализа. Поэтому проектные реализации прецедентов являются намного более детализированными и сложными, чем исходные аналитические реализации прецедентов.

Важно помнить, что моделирование осуществляется лишь для того, чтобы облегчить понимание создаваемой системы. Объем работы должен быть ограничен лишь тем, что на самом деле представляет интерес. Такой подход называют стратегическим проектированием. Существует также тактическое проектирование, которое можно без ущерба перенести в фазу реализации. По сути, полное проектирование осуществляется только тогда, когда предполагается генерировать большую часть кода из модели. И даже в этом случае проектные реализации прецедентов редко активно используются в генерировании кода. По-

этому они создаются, только если необходимо обозначить малопонятные аспекты поведения системы.

20.4. Диаграммы взаимодействий при проектировании

> При проектировании можно уточнять основные диаграммы взаимодействий или создавать новые для иллюстрации центральных механизмов, таких как сохранение объектов.

Диаграммы взаимодействий – основная часть проектной реализации прецедента. Поскольку большие объемы информации проще показать на диаграммах последовательностей, то при проектировании зачастую именно им, а не коммутационным диаграммам, уделяется основное внимание.

Диаграммы взаимодействий в проектировании могут быть:

- уточнением основных аналитических диаграмм взаимодействий с дополнением деталей реализации;
- абсолютно новыми диаграммами, созданными для иллюстрации технических вопросов, возникших при проектировании.

При проектировании вводится ограниченное число центральных механизмов, таких как сохранение объектов, распределение объектов, транзакции и т. д. Часто диаграммы взаимодействий создаются именно для представления этих механизмов. Диаграммы взаимодействий, иллюстрирующие центральные механизмы, нередко охватывают несколько прецедентов.

Чтобы понять роль диаграмм последовательностей в проектировании, рассмотрим прецедент AddCourse, который обсуждался ранее в разделе 12.9.1 (рис. 20.3).

На рис. 20.4 показана аналитическая диаграмма взаимодействий, созданная нами в разделе 12.9.1 (*с. 276*).

На рис. 20.5 представлена обычная диаграмма последовательностей для прецедента AddCourse на ранних этапах проектирования. Как видите, здесь добавлен слой GUI, хотя он и не был смоделирован достаточно глубоко. Также высокоабстрактные операции аналитической диаграммы последовательностей превращены в операции уровня проектирования, достаточно полно описанные для реализации. Например, теперь подробно показано создание объекта посредством вызова явной операции конструктора.

Рисунок 20.5 также включает центральный механизм – обеспечение сохранения объектов Course. В данном случае был выбран очень простой механизм сохранения: :RegistrationManager использует сервисы :DBManager для хранения объектов Course в базе данных. Важно, что этот

Прецедент: AddCourse
ID: 8
Краткое описание: Добавляет детали нового курса в систему.
Главные актеры: Registrar.
Второстепенные актеры: Нет.
Предусловие: 1. Registrar вошел в систему.
Основной поток: 1. Registrar выбирает «add course». 2. Registrar вводит имя нового курса. 3. Система создает новый курс.
Постусловие: 1. Новый курс добавлен в систему.
Альтернативные потоки: CourseAlreadyExists

Рис. 20.3. *Спецификация прецедента AddCourse*

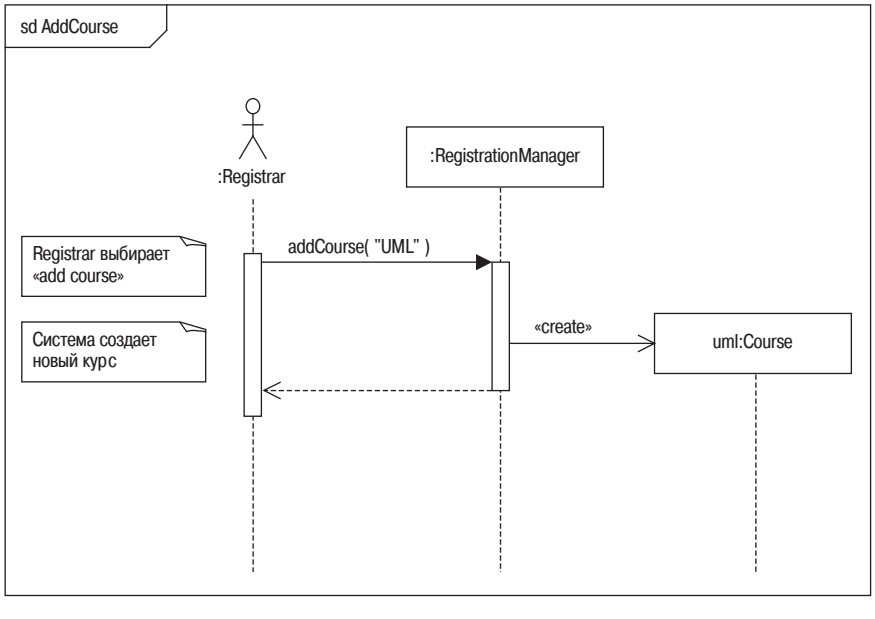


Рис. 20.4. *Аналитическая диаграмма взаимодействий*

центральный механизм, будучи определенным один раз, должен оставаться *неизменным* в течение всего проектирования. Однажды нам пришлось работать над большой системой, в которой было не менее трех разных механизмов сохранения – конечно, это слишком много!

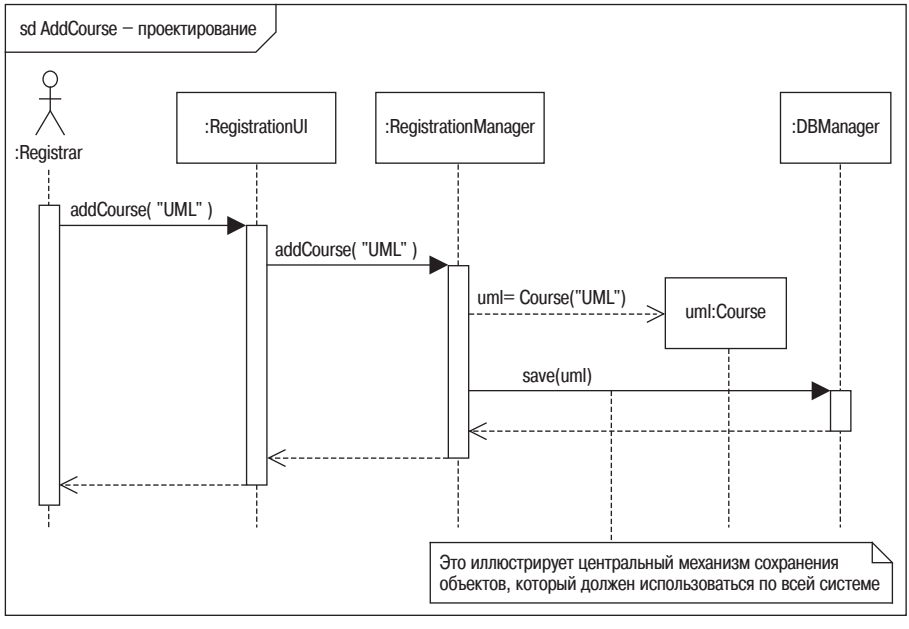


***Рис. 20.5.** Диаграмма последовательностей на ранних этапах проектирования*

20.5. Моделирование параллелизма

> Параллелизм – один из ключевых вопросов, рассматриваемых при проектировании.

Параллелизм означает параллельное выполнение частей системы. Это один из ключевых вопросов, рассматриваемых при проектировании.

UML 2 обеспечивает хорошую поддержку параллелизма:

- активные классы (раздел 20.5.1);
- ветвления и объединения на диаграммах деятельности (раздел 14.8.3);
- оператор par на диаграммах последовательностей (раздел 20.5.2);
- порядковые номера в качестве префиксов на коммуникационных диаграммах (раздел 20.5.3);
- различные представления временных диаграмм (раздел 20.7);
- ортогональные составные состояния на диаграммах состояний (раздел 22.2.2).

В следующем разделе рассматриваются активные классы, а затем обсуждаются параллелизм на диаграммах последовательностей и коммуникационных диаграммах.

20.5.1. Активные классы

> Параллелизм – каждый активный объект имеет собственный поток выполнения.

Основной принцип моделирования параллелизма – каждый поток управления или параллельный процесс моделируется как *активный объект*. Под последним понимают объект, инкапсулирующий собственный поток управления. Активные объекты являются экземплярами активных классов. Активные объекты и активные классы изображаются на диаграммах как обычные классы и объекты, но с двойной рамкой справа и слева, как показано на рис. 20.8.

Параллелизм обычно имеет большое значение для встроенных систем, таких как программное обеспечение, управляющее минифотолабораторией или банкоматом. Для изучения параллелизма рассмотрим очень простую встроенную систему – систему безопасности. Она отслеживает ряд датчиков для обнаружения пожара или проникновения в помещение взломщиков. При срабатывании датчика система включает сигнализацию. Модель прецедентов для системы безопасности показана на рис. 20.6.

Описания прецедентов системы приведены на рис. 20.7. Прецедент ActivateFireOnly (активировать только пожарные датчики) не рассматривается, поскольку основное внимание в этом разделе направлено на аспекты параллелизма системы. Кроме того, эти прецеденты довольно абстрактны и отражают лишь суть того, что должна делать система сигнализации. Более подробно ее поведение будет рассмотрено позже.

Теперь надо найти классы. Для встроенных систем превосходным источником классов могут быть аппаратные средства, на которых система выполняется. На практике наилучшей программной архитектурой обычно является та, которая максимально близка к архитектуре фи-

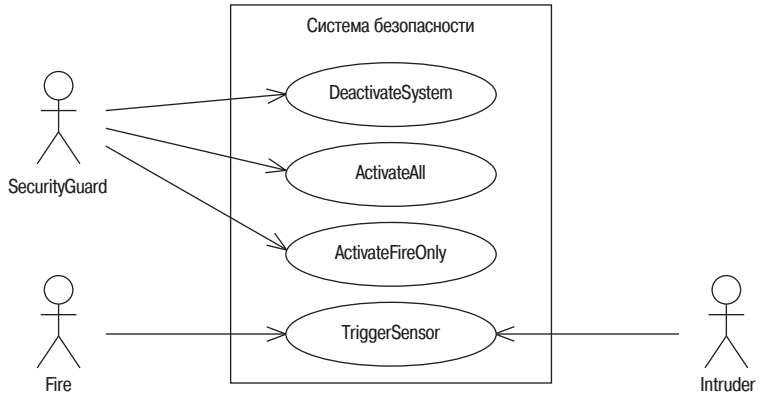


***Рис. 20.6**. Модель прецедентов для системы безопасности*

Прецедент: DeactivateSystem
ID: 1
Краткое описание: Деактивирует систему.
Главные актеры: SecurityGuard
Второстепенные актеры: Нет.
Предусловия: 1. У SecurityGuard есть ключ активации.
Основной поток: 1. SecurityGuard применяет ключ активации для выключения системы. 2. Система прекращает отслеживание датчиков безопасности и пожарных датчиков.
Постусловия: 1. Система безопасности деактивирована. 2. Система безопасности не отслеживает датчики.
Альтернативные потоки: Нет.

Прецедент: ActivateAll (АктивироватьВсе)
ID: 2
Краткое описание: Активирует систему.
Главные актеры: SecurityGuard
Второстепенные актеры: Нет.
Предусловия: 1. У SecurityGuard есть ключ активации.
Основной поток: 1. SecurityGuard применяет ключ активации для включения системы. 2. Система начинает отслеживать датчики безопасности и пожарные датчики. 3. Система воспроизводит звук сирены, демонстрируя готовность.
Постусловия: 1. Система безопасности активирована. 2. Система безопасности отслеживает датчики.
Альтернативные потоки: Нет.

Прецедент: TriggerSensor
ID: 3
Краткое описание: Срабатывает датчик.
Главные актеры: Fire Intruder
Второстепенные актеры: Нет.
Предусловия: 1. Система безопасности активирована.
Основной поток: 1. Если актер Fire инициирует пожарный датчик (FireSensor). 1.1. Сирена воспроизводит сигнал пожарной тревоги. 2. Если актер Intruder инициирует датчик безопасности (SecuritySensor). 2.1. Сирена воспроизводит сигнал тревоги.
Постусловия: 1. Звучит сигнал Сирены.
Альтернативные потоки: Нет.

Рис. 20.7. *Описание прецедентов системы безопасности*

зического оборудования. В рассматриваемом случае сигнализация состоит из четырех компонентов: блока управления, сирены, набора пожарных датчиков и набора датчиков безопасности. Если заглянуть в лок управления, там можно найти плату контроллера для каждого из типов датчиков.

Исходя из прецедентов и информации о физическом оборудовании можно получить диаграмму классов для этой системы, представленную на рис. 20.8.

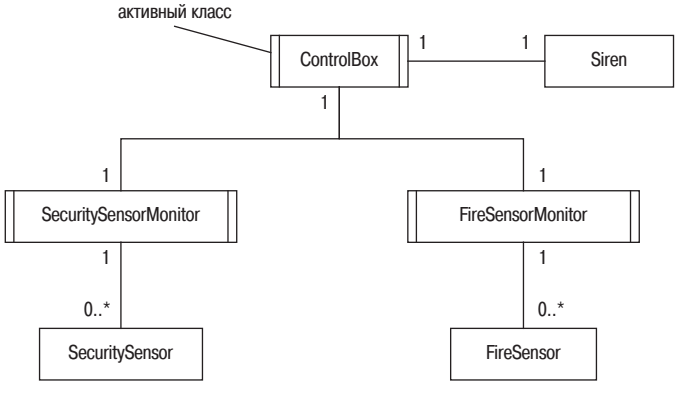


Рис. 20.8. *Диаграмма классов системы безопасности*

> Экземплярами активных классов являются активные объекты.

Система безопасности должна *непрерывно* отслеживать пожарные датчики и датчики безопасности, поэтому необходимо использовать многопоточность (multithreading). Классы ControlBox (блок управления), SecuritySensorMonitor (монитор датчиков безопасности) и FireSensorMonitor (монитор пожарных датчиков) показаны с двойными рамками справа и слева. Это означает, что они являются активными классами.

20.5.2. Параллелизм на диаграммах последовательностей

Теперь мы имеем достаточное количество информации для создания диаграммы последовательностей. Диаграмма последовательностей для прецедента ActivateAll показана на рис. 20.9. Она демонстрирует использование операторов par, loop и critical.

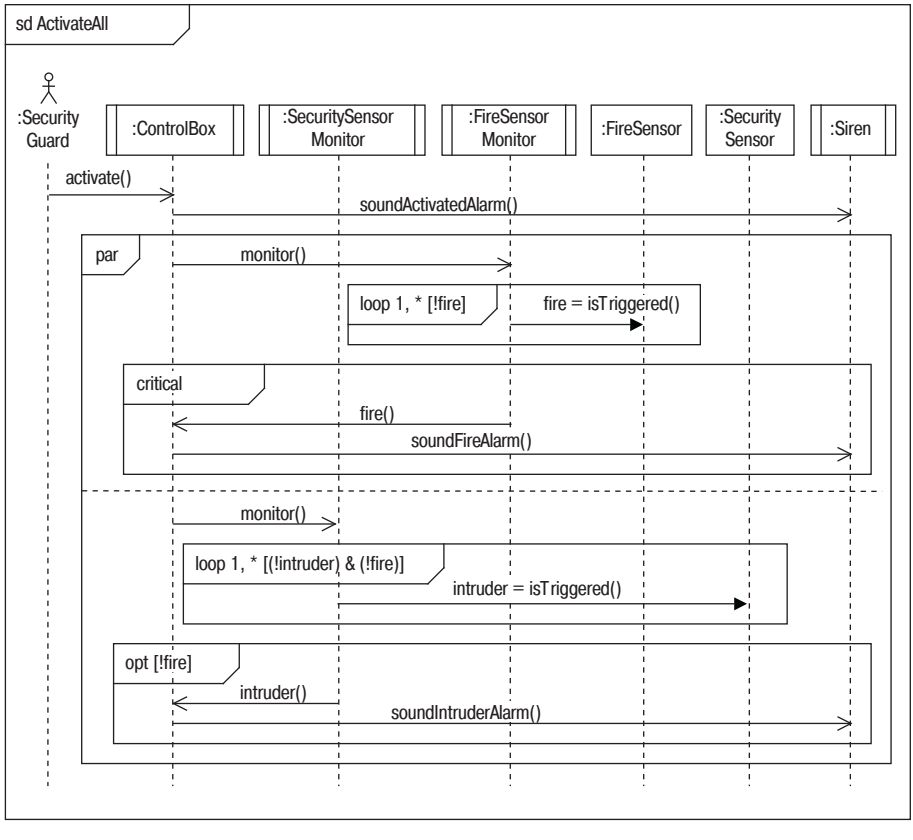


Рис. 20.9. *Диаграмма последовательностей для прецедента ActivateAll*

Ниже приведен поэтапный анализ диаграммы, представленной на рис. 20.9 .

1. Объект :SecurityGuard отправляет сообщение activate() объекту :ControlBox.
2. :ControlBox отправляет сообщение soundActivatedAlarm() объекту :Siren.
3. :ControlBox порождает два потока управления, представленные операндами оператора par. Перемещаясь вниз по диаграмме, сначала вызывается операнд 1 оператора par, а затем операнд 2 оператора par.
4. Операнд 1 оператора par:
   4.1. :ControlBox отправляет сообщение monitor() объекту :FireSensorMonitor.
   4.2. :FireSensorMonitor входит в цикл для опроса :FireSensor. При первом выполнении цикла задается начальное значение переменной fire, затем цикл продолжается до тех пор, пока fire имеет значение false.
   4.3. Когда fire принимает значение true:
      4.3.1. :FireSensorMonitor входит в раздел critical, где:
         4.3.1.1. Он посылает сообщение fire() в :ControlBox.
         4.3.1.2. :ControlBox посылает сообщение soundFireAlarm() объекту :Siren.
5. Операнд 1 оператора par завершается.
6. Операнд 2 оператора par:
   6.1. :ControlBox посылает сообщение monitor() объекту :SecuritySensorMonitor.
   6.2. :SecuritySensorMonitor входит в цикл для опроса :SecuritySensor. При первом выполнении цикла задается начальное значение переменной intruder, затем цикл продолжается до тех пор, пока intruder И (AND) fire имеют значение false.
   6.3. Когда intruder принимает значение true:
      6.3.1. Если fire имеет значение false:
         6.3.1.1. :SecuritySensorMonitor посылает сообщение intruder() объекту :ControlBox;
         6.3.1.2. :ControlBox посылает сообщение soundIntruderAlarm() объекту :Siren.
7. Операнд 2 оператора par завершается.
8. Взаимодействие завершается.

В этом взаимодействии следует отметить несколько интересных моментов:

- Оба операнда par выполняются параллельно.

- Раздел critical представляет атомарное поведение, которое не может быть прервано. Это важное уточнение, поскольку вызов пожарного датчика является критически важным для безопасности и не может быть прерван.
- Оба цикла (loop) имеют семантику Repeat...Until (повторять ... пока) – они выполняются один раз, чтобы задать значение переменной, используемой в их условиях, и затем повторяются до тех пор, пока их условия остаются истинными.
- Пожарная тревога всегда должна иметь бо́льший приоритет по отношению к тревоге вторжения. Поэтому loop в операнде 2 оператора par прерывается событием fire() или intruder(). Зачем продолжать отслеживание вторжений во время пожара! Более того, сигнал тревоги вторжения воспроизводится только в случае отсутствия пожарной тревоги.

Еще хочется отметить, что на этой диаграмме показан только один FireSensor (пожарный датчик) и один SecuritySensor (датчик безопасности). Конечно, этого достаточно для иллюстрации поведения, но, возможно, вам захочется показать итерацию системы по нескольким датчикам SecuritySensor и нескольким FireSensor. Для этого придется изменить диаграмму так, как показано на рис. 20.10, и ввести еще два внутренних цикла для прохода по коллекции датчиков.

Оба операнда на рис. 20.10 с точки зрения цикла ведут себя одинаково, поэтому в качестве примера был выбран верхний из них, который отслеживает FireSensor.

Как видите, внешний цикл остался неизменным. А новый внутренний цикл по очереди перебирает все датчики FireSensor. Это можно показать с помощью выражения цикла:

```
[for each f in FireSensor]
```

Тогда селектор [f] может использоваться для обозначения на диаграмме последовательностей FireSensor, выбранного циклом в качестве линии жизни, и ему может быть отправлено сообщение isTriggered(). В этом цикле есть комбинированный фрагмент break, который прерывает внутренний цикл и выполняет свой операнд, когда fire принимает значение true. При этом, как и ранее, мы попадаем в критическую секцию.

Важно сохранять максимальную простоту диаграмм последовательностей. Основное внимание в реализации прецедента должно быть направлено на иллюстрацию возможных взаимодействий классов при реализации описанного прецедентом поведения. Для данного примера прецедента ActivateAll диаграммы, показанной на рис. 20.9, вероятно, достаточно для представления основного поведения системы, особенно если снабдить ее несколькими комментариями. А рис. 20.10 все-таки слишком подробный.

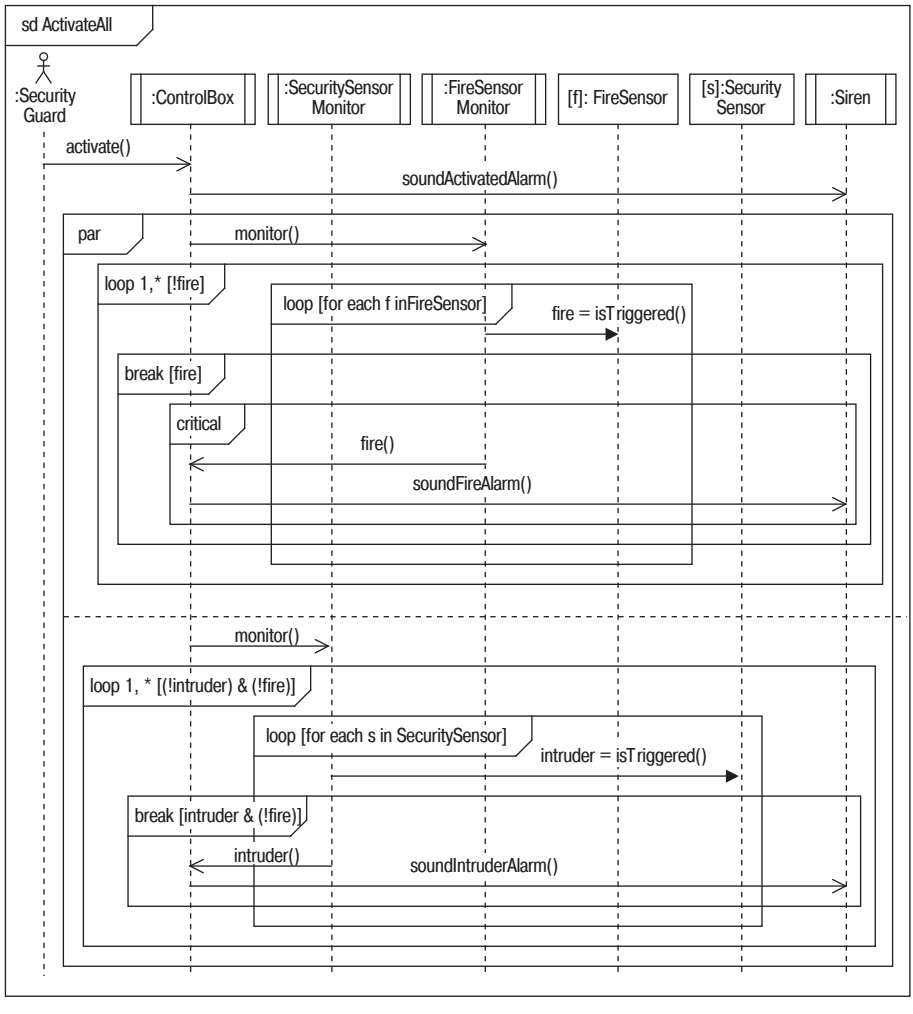


Рис. 20.10. *Диаграмма последовательностей для коллекции датчиков*

20.5.3. Параллелизм на коммуникационных диаграммах

> Для обозначения разных потоков управления после порядкового номера указывается метка потока.

Параллелизм на коммуникационных диаграммах изображают с помощью меток потоков управления, которые указываются после порядкового номера операции, как показано на рис. 20.11. Данная коммуникационная диаграмма представляет то же взаимодействие, что и на рис. 20.9. Здесь есть два параллельных потока управления, A и B.

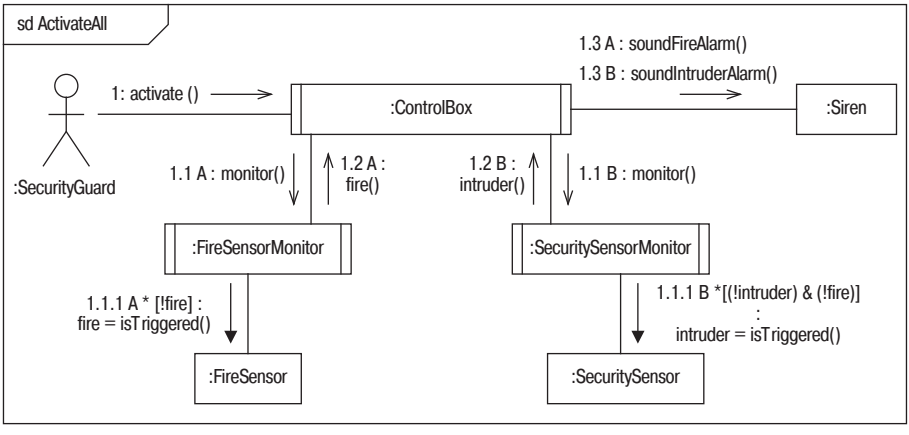


Рис. 20.11. *Коммуникационная диаграмма с двумя параллельными потоками управления*

В этом примере предполагается, что есть всего один экземпляр FireSensor и один экземпляр SecuritySensor и что итерация является операцией последовательного опроса, постоянно вызывающей операцию isTriggered() до тех пор, пока датчик возвращает значение true.

А если бы датчиков было *много*, как предполагалось для диаграммы последовательностей на рис. 20.10? Тогда пришлось бы постоянно проходить набор экземпляров датчиков, опрашивая их по очереди. Для этого необходим вложенный цикл. На коммуникационных диаграммах можно показать вложенные циклы, но нет способа сделать это просто и ясно! В сложных случаях рекомендуется использовать диаграммы последовательностей. Их синтаксис более четкий и гибкий.

20.6. Взаимодействия подсистем

> На диаграммах взаимодействий подсистем можно показать взаимодействия между частями системы.

После создания физической архитектуры подсистем и интерфейсов может оказаться полезным смоделировать взаимодействия между подсистемами. Это обеспечивает очень удобное высокоуровневое представление того, как архитектура реализует прецеденты, без перехода к более низкоуровневым деталям взаимодействий отдельных объектов.

Каждая подсистема рассматривается как черный ящик, который просто предоставляет и требует сервисы, описанные как предоставляемые и требуемые интерфейсы. О взаимодействиях объектов внутри подсистемы вообще не надо беспокоиться.

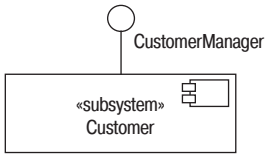


Рис. 20.12. *Подсистема Customer*

На рис. 20.12 показана подсистема Customer с единственным интерфейсом CustomerManager.

Рисунок 20.13 является частью диаграммы последовательностей, представляющей взаимодействие актера с данной подсистемой. Обратите внимание, как изображен интерфейс: в прямоугольнике, примыкающем снизу к пиктограмме подсистемы. Поскольку интерфейс является частью подключаемой подсистемы, он может иметь собственную линию жизни, и сообщения могут направляться непосредственно к этой линии жизни.

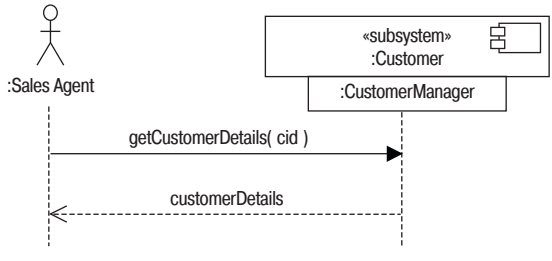


Рис. 20.13. *Диаграмма взаимодействия актера с подсистемой Customer*

20.7. Временные диаграммы

Одним из слабых мест UML 1 было моделирование систем реального времени. Это такие системы, в которых временные соотношения критически важны и события должны следовать друг за другом в рамках определенного временного окна. Мы говорим «временное окно», а не «время», потому что абсолютное время для нас как разработчиков *неприемлемо*. Когда в модели задается время, на самом деле задается время плюс-минус некоторая погрешность, определяемая внешними факторами, такими как точность системных часов. Обычно это не является проблемой, за исключением систем с очень точными временными ограничениями.

> Временные диаграммы моделируют временные ограничения.

В UML 1 временные ограничения можно было обозначать на разных диаграммах, но не было отдельной диаграммы, предназначенной имен-

но для моделирования временных соотношений. UML 2 предоставляет разработчикам моделей систем реального времени временные диаграммы. Это разновидность диаграммы взаимодействий, основное внимание в которой направлено на моделирование временных ограничений. Таким образом, она идеально подходит для моделирования этого аспекта систем реального времени. Временные диаграммы, аналогичные временным диаграммам UML, в течение многих лет успешно используются в электронной промышленности для моделирования временных ограничений электронных схем.

Временные диаграммы очень просты. Время откладывается на горизонтальной оси слева направо. Линии жизни и их состояния (или определенные условия, накладываемые на линии жизни) располагаются вертикально. Переходы между состояниями линий жизни и условиями представляются в виде графика. На рис. 20.14 приведена простая временная диаграмма для класса Siren. Эта диаграмма иллюстрирует, что происходит, когда формируется событие «вторжение», а затем событие «пожар». Конечно, это самый пессимистичный сценарий, но смоделировать его важно, чтобы понять взаимодействие функций обнаружения взломщика и пожара.

Вот последовательный анализ данной временной диаграммы.

t = 0: :Siren находится в состоянии Off (выключен).

t = 10: Происходит событие intruder, и :Siren переходит в состояние SoundingIntruderAlarm (воспроизведение сигнала тревоги вторжения).

t = 25: Сигнал тревоги вторжения может звучать не более 15 минут согласно местным правилам подачи сигналов тревоги. :Siren

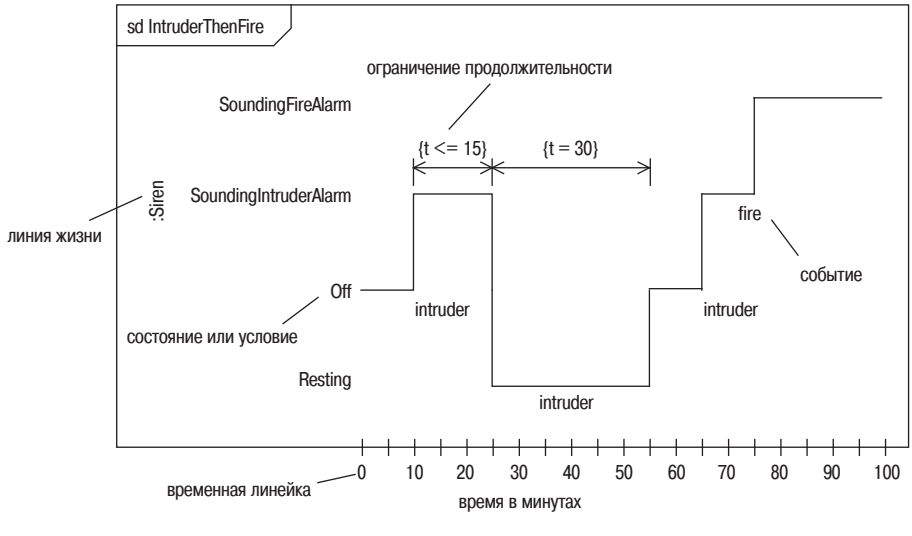


Рис. 20.14. *Временная диаграмма для класса Siren*

переходит в состояние Resting (покой). Он должен находиться в этом состоянии 30 минут (опять же по местным законам).

t = 35: Происходит событие intruder, но :Siren в состоянии Resting, поэтому не может воспроизводить сигнал тревоги.

t = 55: :Siren возвращается в состояние Off.

t = 65: Возникает еще одно событие intruder. :Siren переходит из состояния Off в состояние SoundingIntruderAlarm.

t = 75: Происходит событие fire. :Siren переходит из состояния SoundingIntruderAlarm в состояние SoundingFireAlarm (воспроизведение сигнала пожарной тревоги).

t = 100: Взаимодействие завершается, :Siren остается в состоянии SoundingFireAlarm.

Временные диаграммы также можно представить в более компактной форме, когда состояния располагаются горизонтально. На рис. 20.15 в такой форме показана временная диаграмма с рис. 20.14.

В такой компактной форме акцент обычно смещается больше на состояния и *относительное* время, а не на представление абсолютного времени, как это моделирует временная линейка.

> Временные диаграммы могут использоваться для представления изменений состояния объекта во времени.

Временные диаграммы также могут использоваться для иллюстрации временных ограничений во взаимодействиях между двумя или более линиями жизни. На рис. 20.16 показано взаимодействие между линиями жизни :FireSensorMonitor, :IntruderSensorMonitor и :Siren.

На этой временной диаграмме следует отметить несколько интересных моментов:

- Временнаяй диаграмма имеет три отделения, по одному для каждой линии жизни.

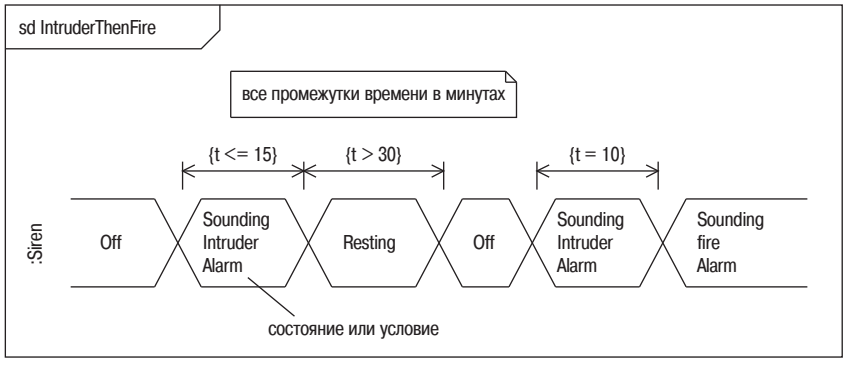


Рис. 20.15. *Компактная форма временной диаграммы с рис. 20.14*

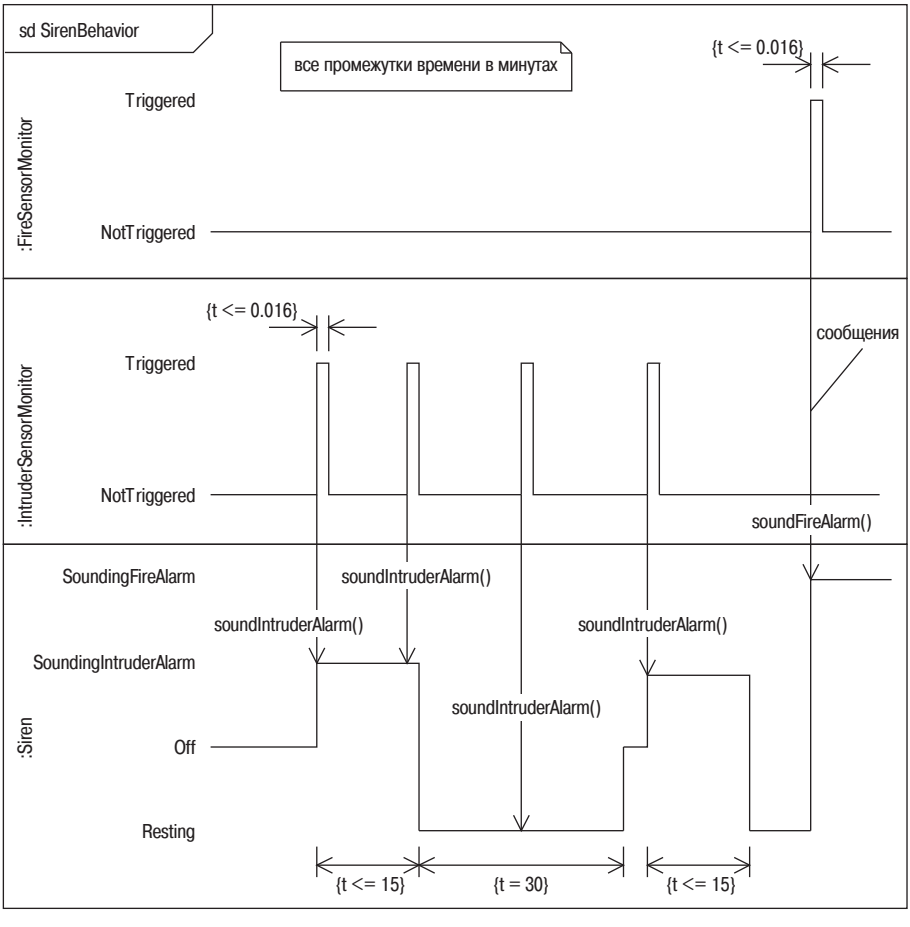


Рис. 20.16. *Временная диаграмма иллюстрирует временные ограничения, накладываемые на взаимодействия между линиями жизни*

- На временных диаграммах можно показать сообщения, которыми обмениваются линии жизни, как представлено на рисунке.
- При вызове датчики обоих типов переходят в состояние Triggered (инициирован), а затем в течение 1 секунды возвращаются в состояние NotTriggered (не инициирован). Это значит, что у обоих датчиков короткое время повторной готовности – важнейшая характеристика.
- :Siren отвечает на события вторжения только в состоянии Off. В состоянии Resting он игнорирует эти события. Это обусловлено местными правилами, согласно которым сигнал тревоги вторжения должен подаваться только в течение 15 минут, а затем выключаться, по крайней мере, на полчаса.

- :Siren *всегда* отвечает на события пожара, даже если находится в состоянии Resting, потому что пожарная сигнализация *должна* включаться как можно скорее после возникновения события пожара.

Как видите, временные диаграммы являются удобным способом моделирования временных ограничений, накладываемых на взаимодействия.

20.8. Пример реализации прецедента на этапе проектирования

В данном разделе рассматривается реальный пример проектирования реализации прецедента. Обсуждаемый пример – это простой редактор прецедентов, ориентированный на схемы. Это часть системы SUMR, которая описывается в приложении B. Ознакомьтесь с этим приложением, прежде чем читать дальше.

Как сказано в приложении B, приложение, которое мы собираемся рассмотреть, является простым редактором прецедентов с синтаксической подсветкой имен актеров, имен прецедентов, включений и расширений. Модель прецедентов для системы UseCaseEditor (редактор прецедентов) показана на рис. 20.17. Она была разработана с помощью инстру-

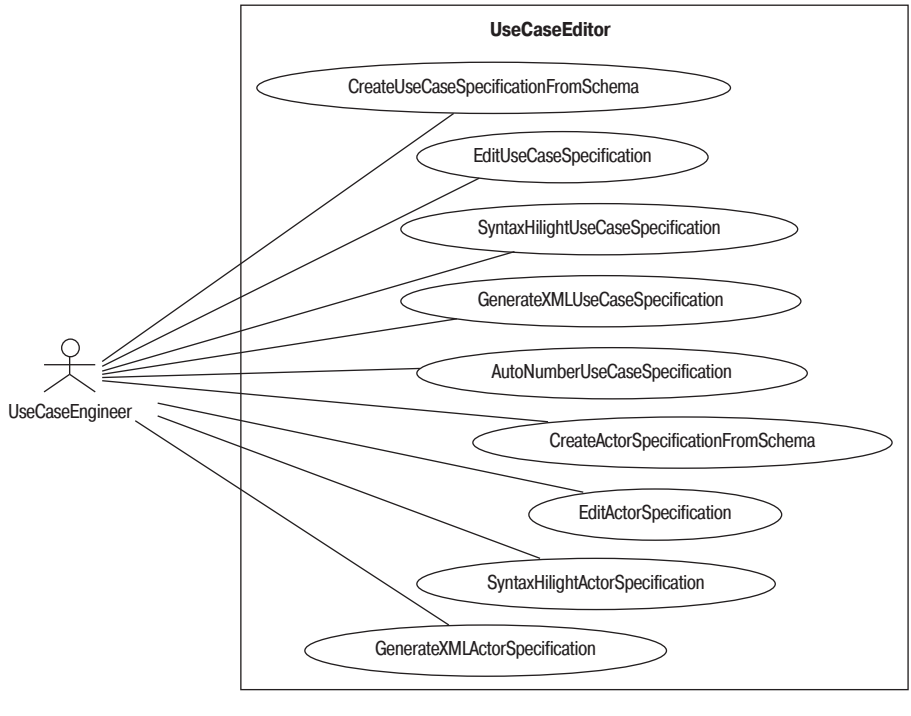


Рис. 20.17. *Модель прецедентов для системы UseCaseEditor*

Прецедент: CreateUseCaseSpecificationFromSchema
ID: 1
Краткое описание: Система создает спецификацию нового прецедента из схемы прецедента.
Главные актеры: UseCaseEngineer
Второстепенные актеры: Нет.
Предусловия: 1. Существует схема прецедента.
Основной поток: 1. Прецедент начинается, когда UseCaseEngineer выбирает «create new use case». 2. Система запрашивает имя прецедента. 3. UseCaseEngineer вводит имя прецедента. 4. Система создает новый прецедент из схемы прецедента. 5. Система представляет новый прецедент для редактирования.
Постусловия: 1. Система создала новый прецедент.
Альтернативные потоки: UseCaseAlreadyExists UseCaseEngineerCancels

Рис. 20.18. *Описание прецедента CreateUseCaseSpecificationFromSchema*

ментального средства моделирования UML MagicDraw (*www.magicdraw.com*), поэтому иллюстрации немного отличаются от всех остальных иллюстраций этой книги.

Основным прецедентом этой системы, вероятно, является CreateUseCaseSpecificationFromSchema (создать спецификацию прецедента по схеме). Он отражает назначение системы – предоставление возможности создания (и редактирования) описаний прецедентов на основании заранее существующей схемы. Прецедент CreateUseCaseSpecificationFromSchema представлен на рис. 20.18.

Поскольку система очень проста, аналитическая модель не уточнялась до очень глубокого уровня, и мы смогли быстро перейти к проектированию. Аналитическая диаграмма классов показана на рис. 20.19. Она отражает наши исходные идеи о том, какие классы будут нужны.

Как часть реализации прецедента на этапе анализа была создана аналитическая диаграмма последовательностей (рис. 20.20). Она иллюстрирует наше предположение о том, как система создает файл нового прецедента из существующего файла схемы.

Проектная модель классов представлена на рис. 20.21. Как видите, аналитическая диаграмма классов сильно отличается от проектной. Как упоминалось, это настолько простая система, что мы очень быстро перешли к проектированию, и основная работа по исследованию была проведена в этом рабочем потоке. Если бы система была более сложной, на определение требований и анализ понадобилось бы больше времени.

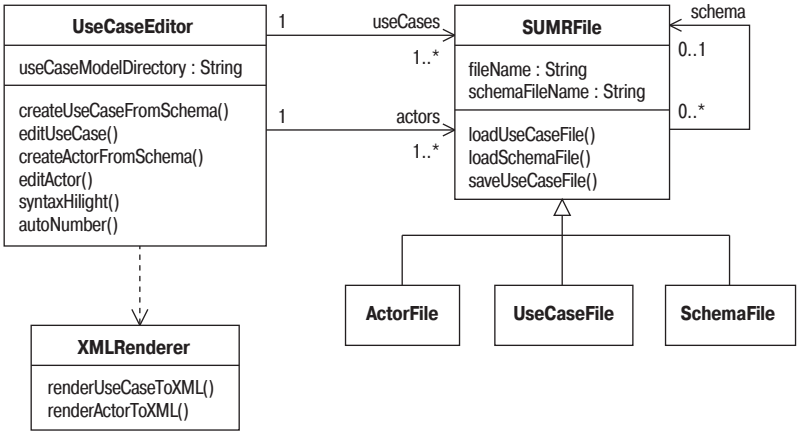


Рис. 20.19. *Аналитическая диаграмма классов*

Редактор прецедентов был исследовательским проектом, и мы не понимали по-настоящему, чем он мог быть полезен, пока не провели несколько итераций и не получили некоторые предварительные исполняемые базовые версии архитектуры, с которыми можно было экспериментировать. Также мы не пытались предугадывать самые эффективные центральные механизмы для файлов SUMR – мы установили их, когда создали первую итерацию системы.

Приложение редактора прецедентов разрабатывалось на языке программирования Python, и все классы, имена которых начинаются с букв wx, являются классами библиотеки GUI wxPython (*www.wxpython.org*).

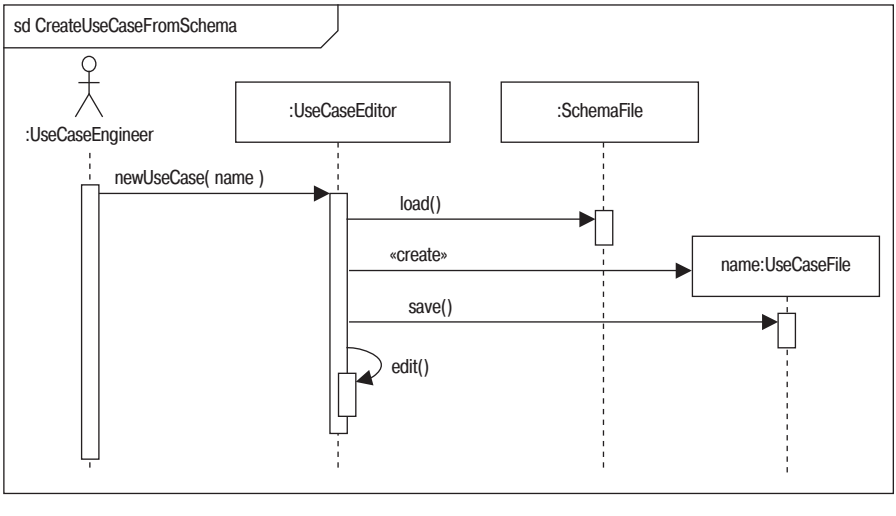


Рис. 20.20. *Аналитическая диаграмма последовательностей*

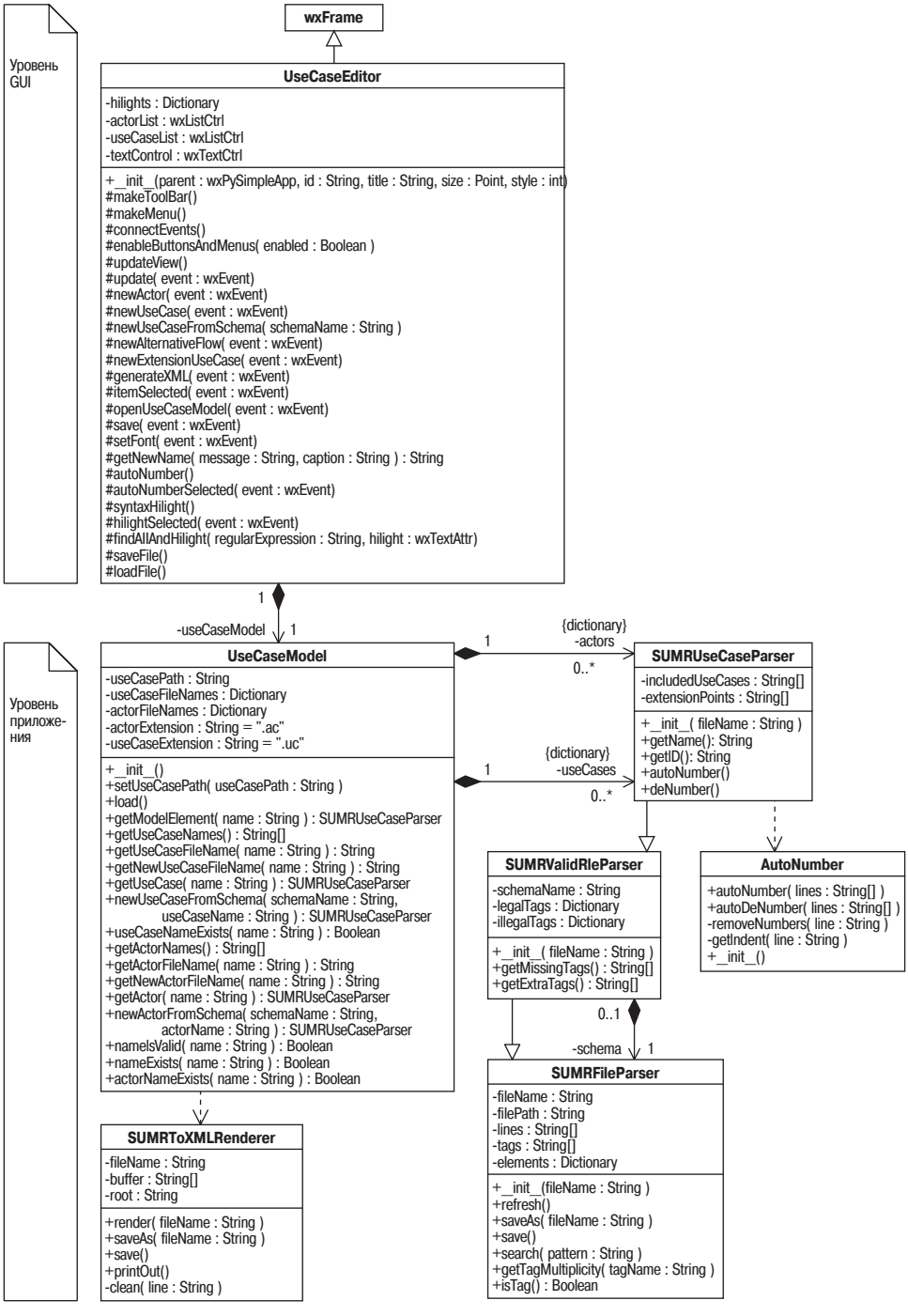


Рис. 20.21. *Проектная модель классов*

Это мощная межплатформенная библиотека GUI, основанная на wxWidgets (*www.wxwidgets.org*). Эти классы придерживаются другого стандарта присваивания имен: начинаются с маленькой буквы, а не как обычно с большой. Разные стандарты наименования в разработке программного обеспечения сложились исторически.

И наконец, давайте посмотрим на диаграмму последовательностей для CreateUseCaseSpecificationFromSchema уровня проектирования (рис. 20.22). Эта диаграмма используется для иллюстрации центральных механизмов создания файла нового прецедента из существующего файла схемы. Эта диаграмма имеет важное значение, потому что данный механизм должен использоваться неизменно во всей системе. Можно убедиться в работоспособности нашего проекта – имеются все необходимые классы и операции для выполнения поставленных задач.

Хотя в этом примере мы представили артефакты определения требований, анализа и проектирования последовательно, необходимо помнить, что UP является итеративным процессом и что наборы этих артефактов на самом деле создаются параллельно. В частности, работа над моделью прецедентов, аналитической диаграммой классов и аналитическими диаграммами взаимодействий будет проводиться одновременно. То же самое относится к проектной диаграмме классов и проектным диаграммам взаимодействий. Обновление артефактов, созданных в предыдущих итерациях, является обычным делом. Помните, что в каждой итерации есть элемент каждого рабочего потока – Определения требований, Анализа, Проектирования, Реализации и Тестирования.

20.9. Что мы узнали

Реализация прецедента на этапе проектирования – это на самом деле расширение реализации прецедента, созданной при анализе. Мы узнали следующее:

- Деятельность UP Проектирование прецедента заключается в выявлении проектных классов, интерфейсов и компонентов, взаимодействие которых обеспечивает поведение, описанное прецедентом.
- Проектные реализации прецедентов – это взаимодействия проектных объектов и классов, направленные на реализацию прецедента. Они включают:
  - проектные диаграммы взаимодействий – уточненные аналитические диаграммы взаимодействий;
  - проектные диаграммы классов – уточненные аналитические диаграммы классов.
- Диаграммы взаимодействий могут использоваться в проектировании для моделирования центральных механизмов, таких как сохранение объектов; эти механизмы могут охватывать многие прецеденты.

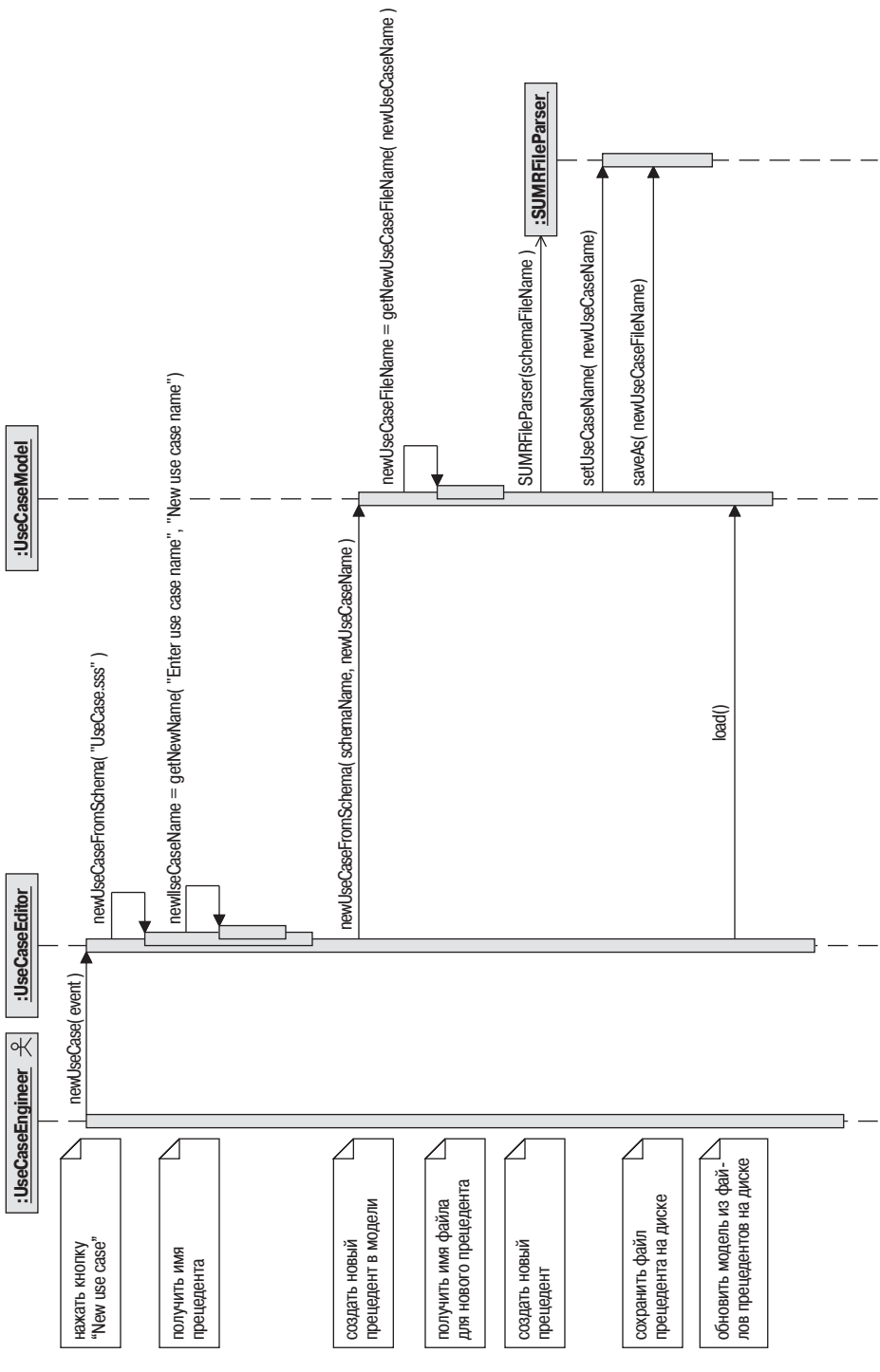


Рис. 20.22. *Диаграмма последовательностей уровня проектирования*

- Моделирование параллелизма.
  - Используются активные классы и объекты.
  - Диаграммы последовательностей:
    - par – все операнды выполняются параллельно;
    - critical – операнд выполняется автоматически без прерываний.
  - Коммуникационные диаграммы:
    - порядковый номер дополняется меткой для обозначения потока управления.
  - Диаграммы деятельности:
    - ветвления;
    - объединения.
- Диаграммы взаимодействий подсистем показывают взаимодействия между разными частями системы на высоком уровне абстракции:
  - в них могут входить актеры, подсистемы, компоненты и классы;
  - части подсистем (например, предоставляемые интерфейсы) могут отображаться в прямоугольниках, примыкающих снизу к пиктограмме подсистемы.
- Временные диаграммы – моделируют временные ограничения:
  - очень полезны для моделирования аппаратных систем реального времени и встроенных систем;
  - время увеличивается вдоль горизонтальной оси слева направо;
  - линии жизни, состояния и условия располагаются по вертикали;
  - переходы между состояниями или условиями изображаются в виде графика;
  - можно показать временные ограничения и события;
  - компактная форма временной диаграммы делает акцент на относительном времени.

21

Конечные автоматы

21.1. План главы

В этой главе обсуждаются конечные автоматы – важное средство моделирования динамического поведения классификаторов.

Глава начинается с введения в конечные автоматы (раздел 21.2), затем обсуждаются два типа конечных автоматов (раздел 21.2.1), автоматы и классы (раздел 21.2.2) и синтаксис конечных автоматов (раздел 21.4). Далее основное внимание уделяется базовым компонентам автоматов: состояниям (раздел 21.5), переходам (раздел 21.6) и событиям (раздел 21.7).

21.2. Конечные автоматы

> Конечный автомат моделирует динамическое поведение реактивного объекта.

И диаграммы деятельности, и диаграммы состояний моделируют аспекты динамического поведения системы, но их семантика и назначение в моделировании сильно различаются. Диаграммы деятельности базируются на технологии сетей Петри (глава 14) и обычно используются для моделирования бизнес-процессов, в которых принимают участие несколько объектов. В основе конечных автоматов UML лежит работа Харела (Harel) [Harel 1]. С их помощью обычно моделируется предыстория жизненного цикла одного реактивного объекта. Она представляется в виде конечного автомата – автомата, который может существовать в конечном числе состояний. В ответ на события конечный автомат четко осуществляет переходы между этими состояниями.

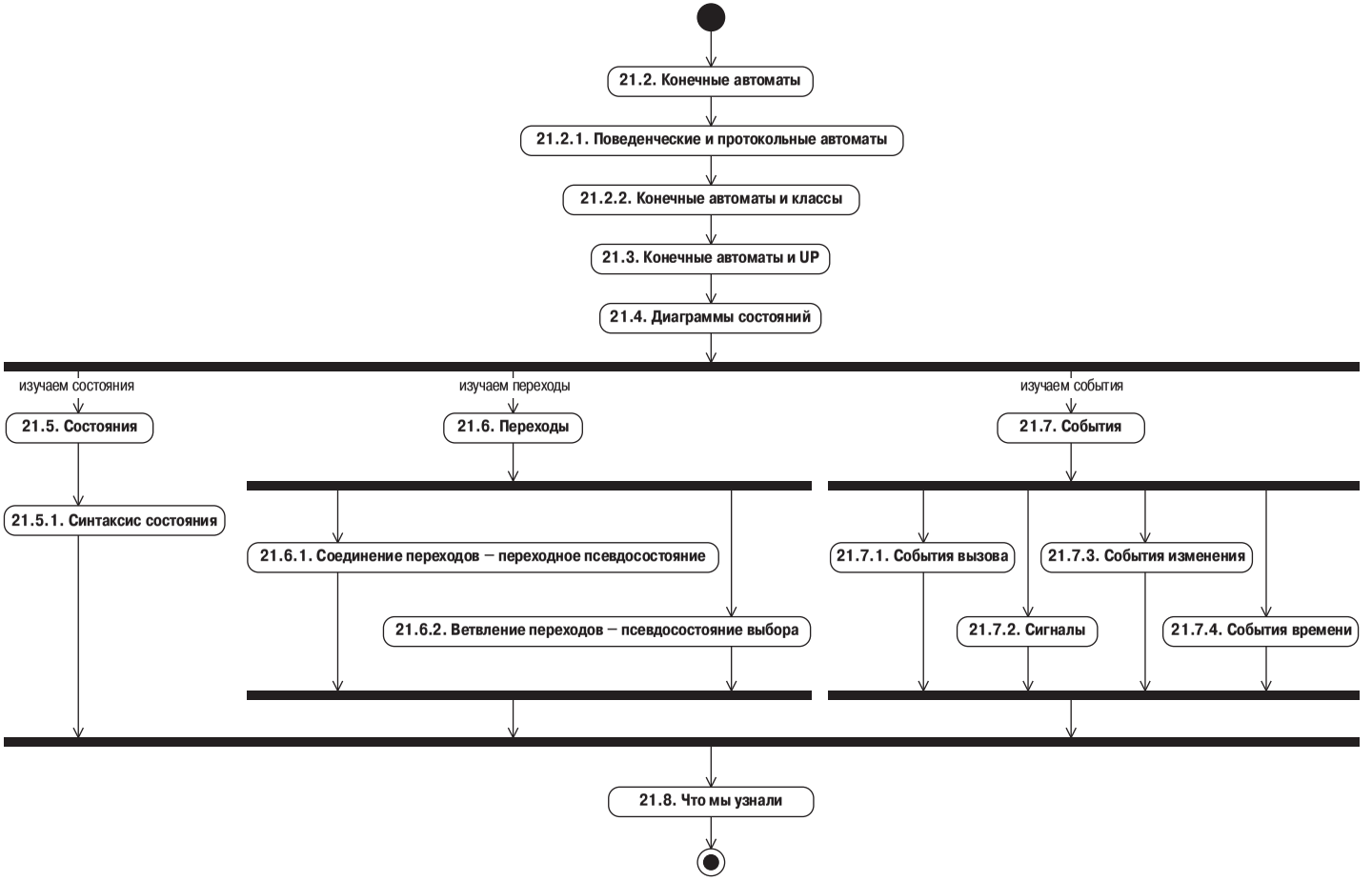


Рис. 21.1. *План главы*

Три основных элемента автоматов – состояния, события и переходы:

- состояние (state) – «условие или ситуация в жизни объекта, при которых он удовлетворяет некоторому условию, осуществляет некоторую деятельность или ожидает некоторого события» [Rumbaugh 1];
- событие (event) – «описание заслуживающего внимания происшествия, занимающего определенное положение во времени и пространстве» [Rumbaugh 1];
- переход (transition) – переход из одного состояния в другое в ответ на событие.

Намного подробнее состояния, события и переходы будут рассмотрены в этой главе несколько позже.

Реактивный объект – это объект, в широком смысле этого слова, который предоставляет автомату контекст. Реактивные объекты:

- отвечают на внешние события (т. е. события, происходящие вне контекста объекта);
- могут генерировать и отвечать на внутренние события;
- их жизненный цикл смоделирован как последовательность состояний, переходов и событий;
- их текущее поведение может зависеть от предыдущего поведения.

Реальный мир полон реактивных объектов, которые можно смоделировать с помощью автоматов. В UML-моделировании конечные автоматы обычно описываются в контексте конкретного классификатора. Затем конечный автомат моделирует поведение, общее для всех экземпляров этого классификатора. Конечные автоматы могут использоваться для моделирования динамического поведения таких классификаторов, как:

- классы;
- прецеденты;
- подсистемы;
- целые системы.

21.2.1. Поведенческие и протокольные автоматы

Спецификация UML 2 определяет два типа конечных автоматов, имеющих общий синтаксис:

- поведенческие автоматы;
- протокольные автоматы.

> Поведенческие автоматы определяют поведение.

Поведенческие автоматы с помощью состояний, переходов и событий определяют *поведение* контекстного классификатора. Они могут использоваться, только если у контекстного классификатора есть некоторое поведение, которое можно смоделировать. У некоторых класси-

фикаторов, например интерфейсов и портов, такого поведения нет; они просто описывают протокол использования. Состояния поведенческих автоматов могут определять одно или более действий, выполняемых при входе в состояние, нахождении в нем или выходе из него (см. раздел 21.5.1).

> Протокольные автоматы определяют протокол.

Протокольные автоматы используют состояния, переходы и события для определения *протокола* контекстного классификатора. Протокол включает:

- условия, при которых могут вызываться операции классификатора и его экземпляров;
- результаты вызовов операций;
- порядок вызовов операций.

Протокольные автоматы ничего не сообщают о реализации этого поведения. Они только определяют, как оно представляется внешней сущности. Протокольные автоматы могут использоваться при описании протокола для *всех* классификаторов, включая те, которые не имеют реализации. Состояния протокольных автоматов *не могут* определять действия; это дело поведенческих автоматов.

На практике разработчики моделей редко проводят различие между поведенческими и протокольными автоматами. Однако по желанию после имени протокольного автомата можно указывать ключевое слово {protocol}.

21.2.2. Конечные автоматы и классы

Конечные автоматы чаще всего используются для моделирования динамического поведения классов. На этом и сосредоточимся.

У каждого класса может быть только один поведенческий автомат, моделирующий все возможные состояния, события и переходы для всех экземпляров этого класса.

У каждого класса также может быть один или более протокольных автоматов, хотя чаще они используются с не имеющими состояния классификаторами, такими как интерфейсы и порты. Класс наследует протокольные автоматы своих родителей.

Если у класса несколько конечных автоматов, все они должны быть совместимы друг с другом.

Поведенческие и протокольные автоматы класса должны определять поведение и протокол, необходимые всем прецедентам, в которых принимают участие экземпляры класса. Если обнаруживается, что прецеденту требуется протокол или поведение, *не* описанное в конечном автомате, это указывает на неполноту автоматов.

21.3. Конечные автоматы и UP

> Конечные автоматы используются преимущественно в рабочем потоке проектирования.

Как и диаграммы деятельностей, автоматы в UP применяются в нескольких рабочих потоках. Они могут использоваться при анализе для моделирования жизненного цикла классов, у которых есть представляющие интерес состояния, например Order и BankAccount. При проектировании с их помощью могут моделироваться такие вещи, как параллелизм и имеющие состояние сеансовые компоненты Java. Мы даже применяли их при определении требований, когда пытались понять сложный прецедент.

Как всегда, главный вопрос: добавит ли что-нибудь существенное в модель создание конечного автомата? Если конечный автомат помогает понять сложный жизненный цикл или поведение, его стоит создать. В противном случае не стоит тратить время на его разработку.

Чаще всего конечные автоматы используются на завершающих этапах фазы Уточнение и в начале фазы Построение, когда осуществляется попытка настолько детально разобраться в классах системы, чтобы можно было их реализовать. Подчас конечные автоматы являются неоценимым подспорьем в детальной разработке системы.

По нашему мнению, самой большой проблемой при использовании конечных автоматов является их тестирование. Рабочий поток тестирования в UP выходит за рамки рассмотрения этой книги. Но здесь все-таки необходимо сказать несколько слов о тестировании конечных автоматов, поскольку этим аспектом тестирования приходится заниматься аналитикам и проектировщикам. Как определить правильность создания конечного автомата? В большинстве инструментальных средств моделирования UML единственная возможность сделать это – вручную провести поэтапный анализ. При этом кто-то должен выступать в роли автомата, чтобы можно было увидеть его реакцию при разных условиях. Обычно лучше всего работать в небольшой группе, где создатель автомата проводит остальных разработчиков модели и экспертов по алгоритму автомата.

Однако *лучший* способ создания и тестирования автоматов – их имитация. Существует несколько инструментов, позволяющих сделать это, например Real-Time Studio от компании Artisan Software (*www.artisansw.com*). С помощью имитации можно выполнить автомат и увидеть его поведение. Некоторые инструментальные средства также позволяют генерировать из автоматов код и тесты. Для моделирования бизнес-систем подобные инструменты излишни, а вот для встроенных систем реального времени, где у объектов могут быть сложное поведение и жизненные циклы, они очень полезны.

21.4. Диаграммы состояний

Для иллюстрации диаграмм состояний (state machine diagrams) давайте рассмотрим простой пример. Один из самых наглядных объектов реального мира, который постоянно переходит из состояния в состояние, – электрическая лампочка. На рис. 21.2 показана передача событий от переключателя к лампочке. Могут быть посланы два события: turnOn (включить) (это событие моделирует подключение лампы в электрическую сеть) и turnOff (выключить) (выключает ток).

Диаграмма состояний содержит только один конечный автомат для единственного реактивного объекта. В данном случае реактивный объект – это система, состоящая из лампочки, переключателя и электропитания. Диаграмма состояний может изображаться в явно обозначенной рамке, как показано на рис. 21.2, или существовать в неявных рамках, предоставляемых средством моделирования.

По желанию имя конечного автомата можно начинать со слов State Machine, но необходимость в этом возникает редко, поскольку автоматы имеют легко опознаваемый синтаксис.

- Состояния обозначаются прямоугольниками со скругленными углами, за исключением начального состояния (закрашенный кружок) и конечного состояния (бычий глаз).
- Переходы указывают на возможные пути между состояниями и моделируются с помощью стрелок.
- События записываются над инициируемыми ими переходами.

Базовая семантика также довольно проста. Когда реактивный объект, находящийся в состоянии A, получает событие anEvent (какое-то событие), он может перейти в состояние B.

У каждого конечного автомата должно быть начальное состояние (закрашенный кружок), обозначающее первое состояние последовательности. Если смена состояний не бесконечна, должно присутствовать и конечное состояние (бычий глаз), которое завершает последовательность переходов. Обычно переход от начального псевдосостояния к первому «настоящему» состоянию происходит автоматически. Начальное псевдосостояние используется просто как удобный маркер для обозначения начала ряда переходов состояний.

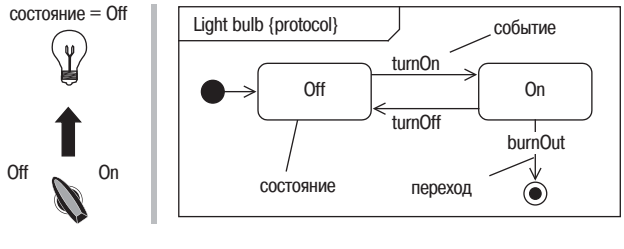


Рис. 21.2. *Диаграмма состояний электрической лампочки*

Когда переключатель устанавливается в положение «On» (включен), лампочке посылается событие turnOn (рис. 21.2). В диаграммах состояний события считаются мгновенными, т. е. от переключателя к лампочке событие доходит мгновенно. Мгновенные события существенно упрощают теорию автоматов. Если бы события не были мгновенными, возникали бы условия состязания, когда два события наперегонки устремляются от источника к одному и тому же реактивному объекту. Это состязание пришлось бы моделировать как некую разновидность автомата!

> События обуславливают переходы состояний.

Лампочка получает событие turnOn и в ответ на него меняет свое состояние на On. В этом суть автоматов – объекты могут менять состояние при получении события. Лампочка переходит в состояние Off при получении события turnOff. В некоторый момент может произойти событие burnOut (лампочка перегорает). Оно завершает конечный автомат.

Все элементы конечного автомата подробно рассматриваются в следующих разделах.

21.5. Состояния

В книге «UML Reference Manual» [Rumbaugh 1] состояние определяется как «условие или ситуация в жизни объекта, при которых он удовлетворяет некоторому условию, осуществляет некоторую деятельность или ожидает некоторого события». Состояние объекта меняется со временем, но в любой отдельный момент оно определяется:

- значениями атрибутов объекта;
- отношениями с другими объектами;
- осуществляемыми деятельностями.

> Состояние – это семантически значимое состояние объекта.

С течением времени объекты обмениваются сообщениями. Эти сообщения и являются событиями, которые могут привести к изменению состояния объекта. Важно очень точно осознать, что мы понимаем под «состоянием». В случае электрической лампочки можно было бы предположить (если бы мы были специалистами в квантовой физике), что каждое изменение любого атома или мельчайшей частицы лампочки образует новое состояние. Строго говоря, это верно, но привело бы нас к несметному числу состояний, по большей части, идентичных.

> Должны быть выявлены значимые состояния системы.

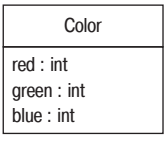


Рис. 21.3. *Класс Color*

Однако с точки зрения пользователя значимыми состояниями лампочки являются On, Off и конечное состояние, когда она перегорает. Выявление *значимых* состояний системы – ключ к успешному моделированию состояний.

В качестве другого примера рассмотрим простой класс Color, приведенный на рис. 21.3.

Если предположить, что атрибуты red (красный), green (зеленый) и blue (синий) могут принимать значения от 0 до 255, тогда на основании только этих значений у объектов данного класса может быть $256 \times 256 \times 256 = 16777216$ возможных состояний. Вот это была бы диаграмма состояний! Однако мы должны задать себе вопрос: в чем основная семантическая разница между каждым из этих состояний? Ответ: никакой разницы. Каждое из 16777216 возможных состояний представляет цвет и только. В сущности, автомат этого класса совсем неинтересен, поскольку все возможные состояния могут быть смоделированы одним-единственным состоянием.

То есть предпосылкой моделирования в виде автомата должно быть наличие семантики «отличия, которое различает» состояния. Конечные автоматы должны повышать ценность модели. Примеры таких автоматов будут даны в этой и следующей главах.

21.5.1. Синтаксис состояния

UML-синтаксис состояния представлен на рис. 21.4.

Каждое состояние поведенческого автомата может содержать нуль или более действий и деятельностей. У состояний протокольных автоматов *нет* действий или деятельностей.

> Действия являются мгновенными и непрерывными.

Действия считаются мгновенными и непрерываемыми, тогда как деятельности занимают конечное время и могут быть прерваны. Каждое действие в состоянии ассоциируется с внутренним переходом, инициируемым событием. В состоянии может быть любое число действий и внутренних переходов.

Внутренний переход позволяет зафиксировать тот факт, что произошло что-то, заслуживающее отражения в модели, но не обуславливаю-

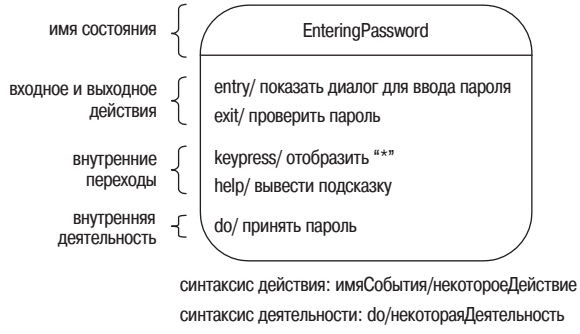


***Рис. 21.4.** Синтаксис состояния*

щее (или не настолько важное, чтобы моделировать это как) переход в новое состояние. Например, на рис. 21.4 нажатие одной из клавиш клавиатуры, конечно, является заслуживающим внимания событием, но оно не приводит к переходу из состояния EnteringPassword (ввод пароля). Мы моделируем это как внутреннее событие keypress (нажатие клавиши), которое обусловливает внутренний переход, инициирующий действие отобразить "*".

Два специальных действия – вход и выход – ассоциированы со специальными событиями entry и exit. У этих двух событий особая семантика. Событие entry происходит мгновенно и автоматически при входе в состояние. Это первое, что происходит, когда осуществляется вход в состояние. Это событие обусловливает выполнение ассоциированного с ним действия на входе. Событие exit – самое последнее, что происходит мгновенно и автоматически при выходе из состояния. Обусловливает выполнение ассоциированного действия на выходе.

> Деятельности длятся конечный промежуток времени и могут быть прерваны.

Деятельности, с другой стороны, длятся конечный промежуток времени и могут быть прерваны по получении события. Деятельность обозначается ключевым словом do (делать). В отличие от действий, которые всегда завершаются, потому что они атомарны, деятельность можно прервать до того, как завершилось ее выполнение.

21.6. Переходы

UML-синтаксис переходов для поведенческих автоматов представлен на рис. 21.5.

> Переходы показывают движение между состояниями.

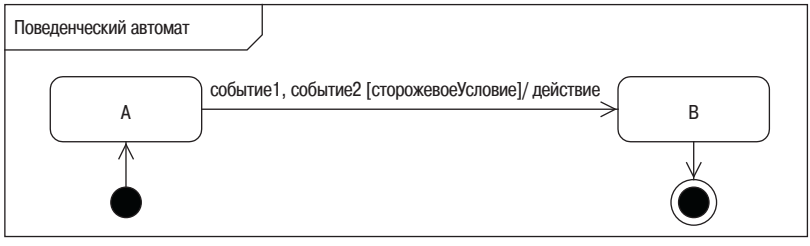


Рис. 21.5. *Синтаксис переходов для поведенческих автоматов*

Синтаксис переходов в поведенческом автомате прост и может использоваться для внешних переходов (изображаются стрелками) или внутренних переходов (изображаются вложенными в состояние). Каждый переход имеет три необязательных элемента.

1. Нуль или более событий – определяют внешнее или внутреннее происшествие, которое может инициировать переход.
2. Нуль или одно сторожевое условие – логическое выражение, которое должно быть выполнено (иметь значение true), прежде чем может произойти переход. Условие указывают после событий.
3. Нуль или более действий – часть работы, ассоциированная с переходом и выполняемая при срабатывании перехода.

Рисунок 21.5 можно прочитать так: «При возникновении (события1 ИЛИ события2), если (сторожевоеУсловие является истинным), выполнить действие и немедленно перейти в состояние B».

В действиях могут использоваться переменные, область действия которых – конечный автомат. Например:

```
actionPerformed( actionEvent )/ command = actionEvent.getActionCommand()
```

В приведенном примере actionPerformed(actionEvent) – событие, генерируемое нажатием кнопки в Java GUI. При получении этого события выполняется действие по сохранению имени кнопки в переменной command.

Синтаксис переходов протокольных автоматов немного другой, как показано на рис. 21.6.

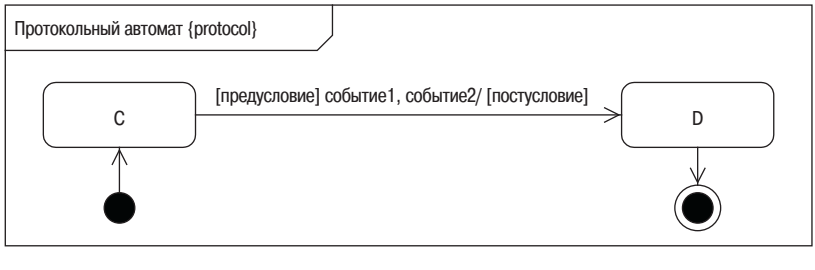


Рис. 21.6. *Синтаксис переходов для протокольных автоматов*

- Здесь нет действия, поскольку задается протокол, а не реализация.
- Сторожевое условие заменено предусловиями и постусловиями. Обратите внимание, что предусловие указывается *перед* событиями, а постусловие – после слэша.

Как в поведенческих, так и в протокольных автоматах переход, осуществляемый без события, называют *автоматическим переходом* (*automatic transition*). Автоматический переход не ожидает события и срабатывает при выполнении его сторожевого условия или предусловия.

21.6.1. Соединение переходов – переходное псевдосостояние

> Переходные псевдосостояния объединяют или разветвляют переходы.

Переходы могут быть соединены *переходными псевдосостояниями* (*junction pseudo-states*). Это точки слияния или ветвления переходов. Они изображаются в виде закрашенных кружков с одним или более входными переходами и одним или более исходящими переходами. Пример на рис. 21.7 показывает конечный автомат для класса Loan, который был представлен в разделе 18.12.2. Loan моделирует получение книги в библиотеке. Конечный автомат Loan имеет простой узел слияния. Это наиболее распространенное использование переходных псевдосостояний.

У переходного псевдосостояния может быть несколько выходных переходов. В этом случае каждый исходящий переход должен быть защищен взаимоисключающим сторожевым условием, обеспечивающим возможность срабатывания только одного перехода. Пример представлен на рис. 21.8. Здесь конечный автомат для класса Loan был расширен, чтобы обработать случай, когда выдача книги может быть продлена. Бизнес-правило состоит в том, что для продления срока возврата выданная книга должна быть предъявлена библиотеке. Таким образом, события returnBook по-прежнему действительны.

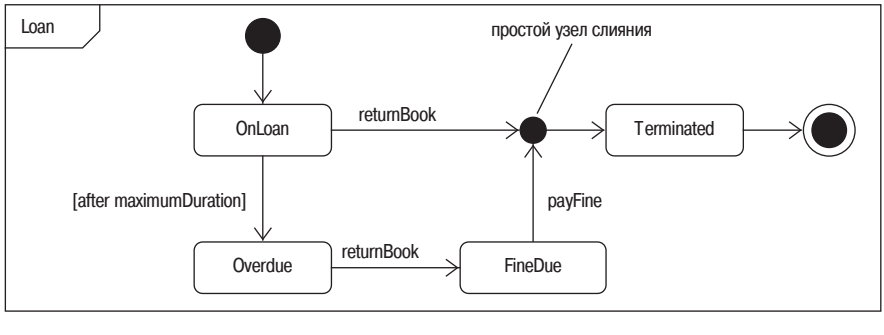


Рис. 21.7. *Конечный автомат класса Loan имеет простой узел слияния*

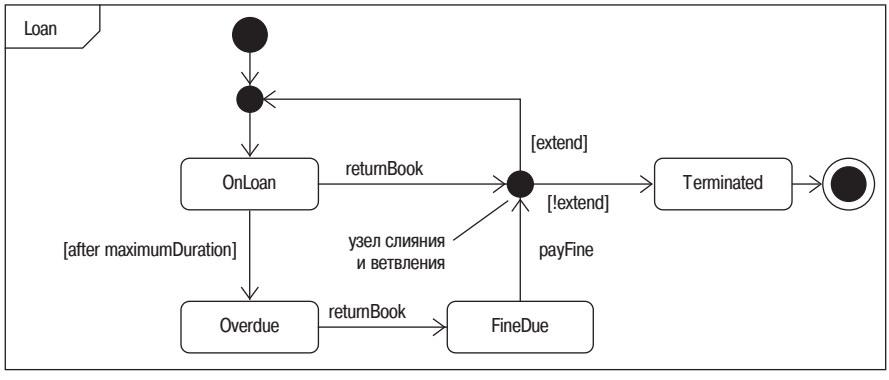


Рис. 21.8. *Конечный автомат класса Loan имеет узел слияния и ветвления*

21.6.2. Ветвление переходов – псевдосостояние выбора

Для представления простого ветвления без слияния необходимо использовать *псевдосостояние выбора* (*choice pseudo-state*), как показано на рис. 21.9.

> Псевдосостояние выбора направляет поток конечного автомата согласно заданным условиям.

Псевдосостояние выбора позволяет направлять поток конечного автомата согласно условиям, заданным для исходящих переходов. Напри-

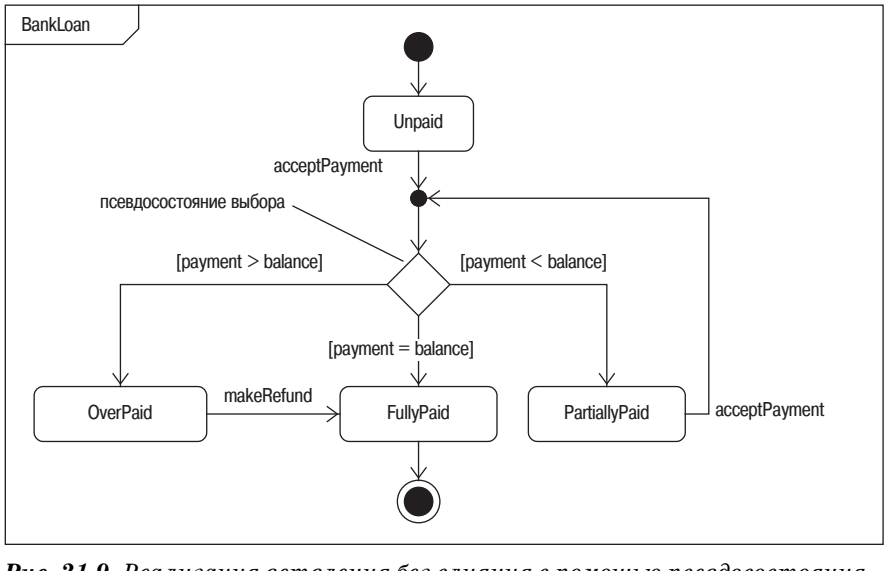


Рис. 21.9. *Реализация ветвления без слияния с помощью псевдосостояния выбора*

мер, на рис. 21.9 показан поведенческий автомат простого класса BankLoan (банковская ссуда). При получении события acceptPayment (принят платеж) объект BankLoan переходит из состояния Unpaid (не оплачен) в одно из трех состояний – FullyPaid (полностью оплачен), OverPaid (переплачен) или PartiallyPaid (частично оплачен) – в зависимости от соотношения объема платежа (payment) и неоплаченного баланса BankLoan.

Условия, налагаемые на исходящие переходы псевдосостояния выбора, должны быть взаимоисключающими, чтобы гарантировать в любой момент времени срабатывание только одного из них.

21.7. События

> События инициируют переходы.

UML определяет событие как «описание заслуживающего внимания происшествия, занимающего определенное положение во времени и пространстве». События инициируют переходы в автоматах. События могут быть указаны вне состояний на переходах, как на рис. 21.10, или внутри состояний, как на рис. 21.11.

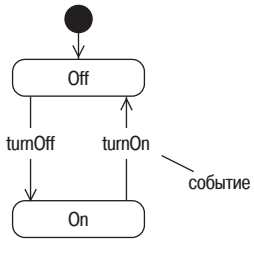


***Рис. 21.10.** События указаны вне состояний на переходах*

Существует четыре семантически различных типа событий:

- событие вызова;
- сигнал;
- событие изменения;
- событие времени.

21.7.1. События вызова

Событие вызова (call event) – это запрос на вызов конкретной операции экземпляра контекстного класса.

> Событие вызова – это запрос на вызов конкретной операции.

Сигнатура события вызова должна быть аналогичной сигнатуре операции контекстного класса автомата. Получение события вызова – это

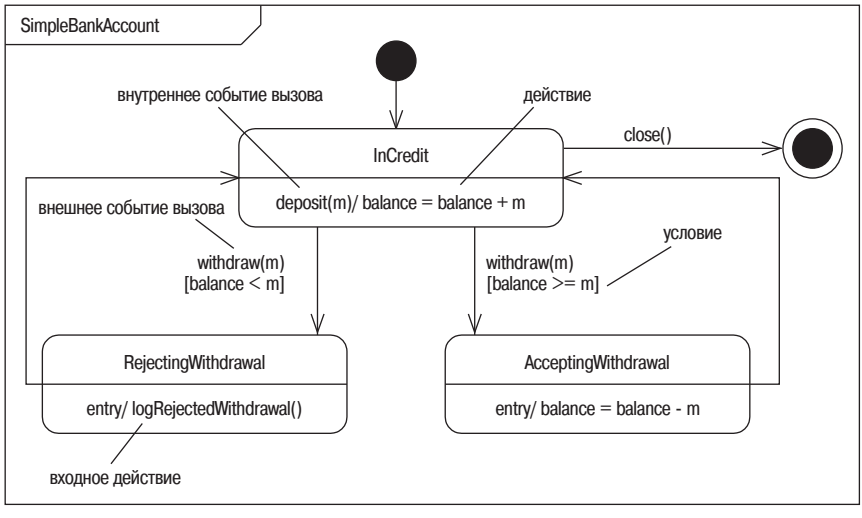


Рис. 21.11. *Внутренние и внешние события вызова*

триггер на выполнение операции. Таким образом, событие вызова – вероятно, самый простой тип событий.

В примере на рис. 21.11 показан фрагмент конечного автомата класса SimpleBankAccount (простой банковский счет). На данный класс наложены следующие ограничения:

- баланс счетов всегда должен оставаться бо́льшим или равным нулю;
- запрос на снятие денег будет отклонен, если запрашиваемая сумма превышает баланс.

На рисунке показаны внутренние и внешние события вызова. Они соответствуют операциям класса SimpleBankAccount.

Для события вызова может быть задана последовательность действий, в которой действия перечисляются через точку с запятой. Они определяют семантику операции и могут использовать атрибуты и операции контекстного класса. Если у операции есть возвращаемый тип, событие вызова имеет возвращаемое значение этого типа.

21.7.2. Сигналы

> Сигнал – это однонаправленная асинхронная связь между объектами.

Сигнал – это пакет информации, асинхронно передаваемый между объектами. Сигнал моделируется как отмеченный стереотипом класс, содержащий в своих атрибутах информацию, которая должна быть передана (рис. 21.12). У сигнала обычно нет операций, потому что он предназначен только для передачи информации.

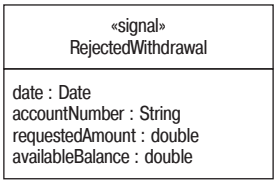


Рис. 21.12. *Сигнал – это класс, отмеченный стереотипом «signal»*

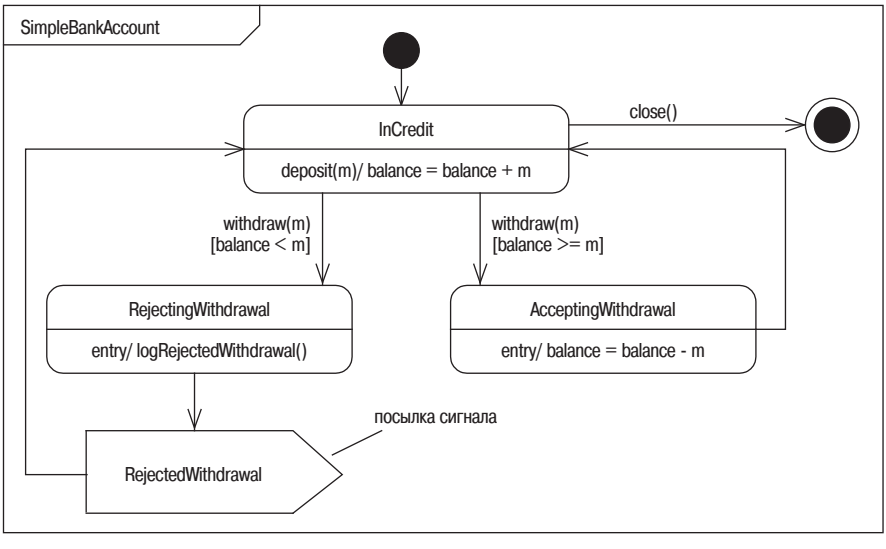


Рис. 21.13. *Посылка сигнала*

На рис. 21.13 представлен скорректированный конечный автомат для SimpleBankAccount. Теперь при отклонении запроса на снятие денег со счета посылается сигнал. Передача сигнала обозначается выпуклым пятиугольником, внутри которого указывается имя сигнала. Такой же синтаксис используется на диаграммах деятельностей (см. раздел 15.6, где сигналы обсуждаются более подробно).

Прием сигнала обозначается вогнутым пятиугольником, как показано во фрагменте конечного автомата на рис. 21.14. В нем указана операция контекстного класса, которая принимает сигнал как параметр.

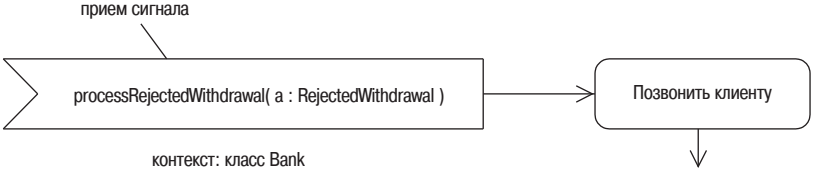


Рис. 21.14. *Прием сигнала*

Прием сигнала также можно показать на внешних или внутренних переходах с помощью стандартной нотации имяСобытия/действие, которая обсуждалась в разделах 21.5.1 и 21.6.

21.7.3. События изменения

> События изменения имеют место, когда значение логического выражения меняется с false на true.

Событие изменения задается в виде логического выражения, как показано на рис. 21.15. Ассоциированное с событием действие выполняется, когда логическое выражение меняет значение с false на true. Все значения логического выражения должны быть константами, глобальными переменными или атрибутами и операциями контекстного класса. С точки зрения реализации событие изменения означает постоянную проверку логического условия при нахождении в определенном состоянии.

На рис. 21.15 конечный автомат SimpleBankAccount был изменен таким образом, что менеджер получает уведомление, если баланс счета становится равным или большим 5000. Это уведомление необходимо для того, чтобы менеджер мог предупредить клиента о других вариантах вклада.

События изменения являются срабатывающими по *положительному фронту* (*positive edge triggered*). Это означает, что они инициируются при каждом изменении значения логического выражения с false на true.

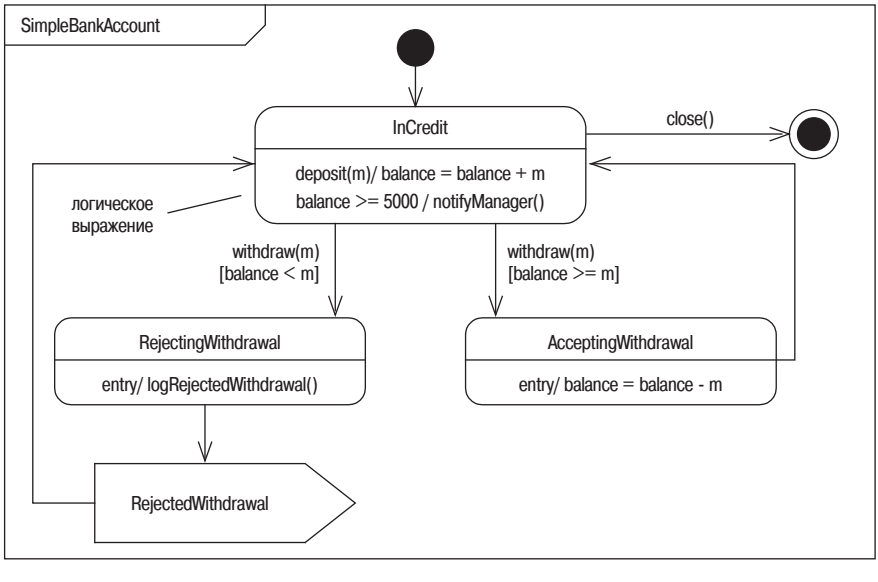


Рис. 21.15. *Событие изменения в виде логического выражения*

Чтобы событие изменения было инициировано, логическое выражение должно вернуться к значению false, а затем снова перейти к true.

Срабатывание по положительному фронту – это именно то поведение, которое требуется для класса SimpleBankAccount. Действие notifyManager() (уведомить менеджера) будет инициироваться, *только* когда баланс счета достигнет 5000 или превысит эту сумму. Понятно, что если баланс будет колебаться вокруг 5000, менеджер получит множество уведомлений. Мы предполагаем, что у менеджера есть некоторый бизнес-процесс для обработки такой ситуации.

21.7.4. События времени

> События времени происходят в ответ на время.

События времени обычно обозначаются ключевыми словами when (когда) и after (после). Ключевое слово when определяет *конкретное* время, когда событие инициируется; after определяет порог времени, *после* которого событие инициируется. Например: when(date = 07/10/2005) и after(3 months).

На диаграмме для каждого события времени обязательно должны быть указаны единицы измерения времени (часы, дни, месяцы и т. д.). Все символы выражения должны быть константами, глобальными переменными или атрибутами и операциями контекстного класса.

Пример на рис. 21.16 – это фрагмент автомата класса CreditAccount, имеющего ограниченные (несколько суровые) условия получения кредита. Как видите, если объект CreditAccount находится в состоянии Overdrawn (кредит превышен) в течение трех месяцев, он переходит в состояние Frozen (заморожен).

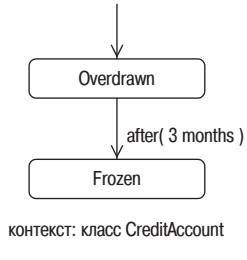


***Рис. 21.16.** Фрагмент автомата класса CreditAccount с ограничением по времени*

21.8. Что мы узнали

В этой главе обсуждалось построение простых конечных автоматов с помощью состояний, действий, деятельностей, событий и переходов. Мы узнали следующее:

- Конечные автоматы основаны на работе Харела.
  - Конечные автоматы моделируют динамическое поведение реактивного объекта.
  - Диаграммы состояний содержат единственный конечный автомат одного реактивного объекта.
- Реактивный объект обеспечивает контекст конечного автомата.
  - Реактивные объекты:
    - отвечают на внешние события;
    - могут генерировать внутренние события;
    - имеют четкий жизненный цикл, который может быть смоделирован как последовательность состояний, переходов и событий;
    - имеют текущее поведение, которое может зависеть от предыдущего.
  - Примеры реактивных объектов:
    - классы (самые распространенные);
    - прецеденты;
    - подсистемы;
    - целые системы.
- Существует два типа конечных автоматов:
  - поведенческие автоматы:
    - моделируют поведение контекстного классификатора;
    - состояния в поведенческих автоматах могут содержать нуль или более действий и деятельностей;
  - протокольные автоматы:
    - моделируют протокол контекстного классификатора;
    - состояния в протокольных автоматах не имеют действий и деятельностей.
- Действия – мгновенные и непрерываемые части работы:
  - могут иметь место в состоянии и быть ассоциированными с внутренним переходом;
  - могут иметь место вне состояния и быть ассоциированными с внешним переходом.
- Деятельности – части работы, которые занимают конечный промежуток времени и могут быть прерваны:
  - могут иметь место только в состоянии.
- Состояние – семантически значимое состояние объекта.
  - Состояние объекта определяется:
    - значениями атрибутов объекта;
    - отношениями с другими объектами;

    - деятельностями, осуществляемыми объектом.
  - Синтаксис состояния:
    - входное действие – осуществляется немедленно при входе в состояние;
    - выходное действие – осуществляется немедленно при выходе из состояния;
    - внутренние переходы – обусловливаются событиями, недостаточно существенными для инициации перехода в новое состояние:
      - событие обрабатывается внутренним переходом в рамках состояния;
    - внутренняя деятельность – часть работы, которая занимает конечный промежуток времени и может быть прервана.
- Переход – перемещение между двумя состояниями.
  - Синтаксис перехода:
    - событие – событие, инициирующее переход;
    - сторожевое условие – логическое выражение, которое должно выполниться, чтобы переход произошел;
    - действие – действие, которое происходит вместе с переходом;
    - переходное псевдосостояние – объединяет или разветвляет переходы;
    - псевдосостояние выбора – направляет поток конечного автомата согласно наложенным условиям.
- Событие – что-то существенное, происходящее с реактивным объектом. Типы событий:
  - событие вызова:
    - вызов необходимого набора действий;
    - вызов операции объекта;
  - сигнал:
    - прием сигнала – сигнал это асинхронная однонаправленная связь между объектами;
  - событие изменения:
    - имеет место, когда некоторое логическое условие меняет свое значение с false на true (т. е. ребро инициируется при переходе ложь-истина);
  - событие времени:
    - ключевое слово after – происходит по прошествии некоторого промежутка времени;
    - ключевое слово when – происходит, когда выполняется некоторое временное условие.

22

Дополнительные аспекты конечных автоматов

22.1. План главы

Эту главу мы начинаем с обсуждения составных состояний. Это состояния, содержащие вложенный конечный автомат. В разделе 22.2

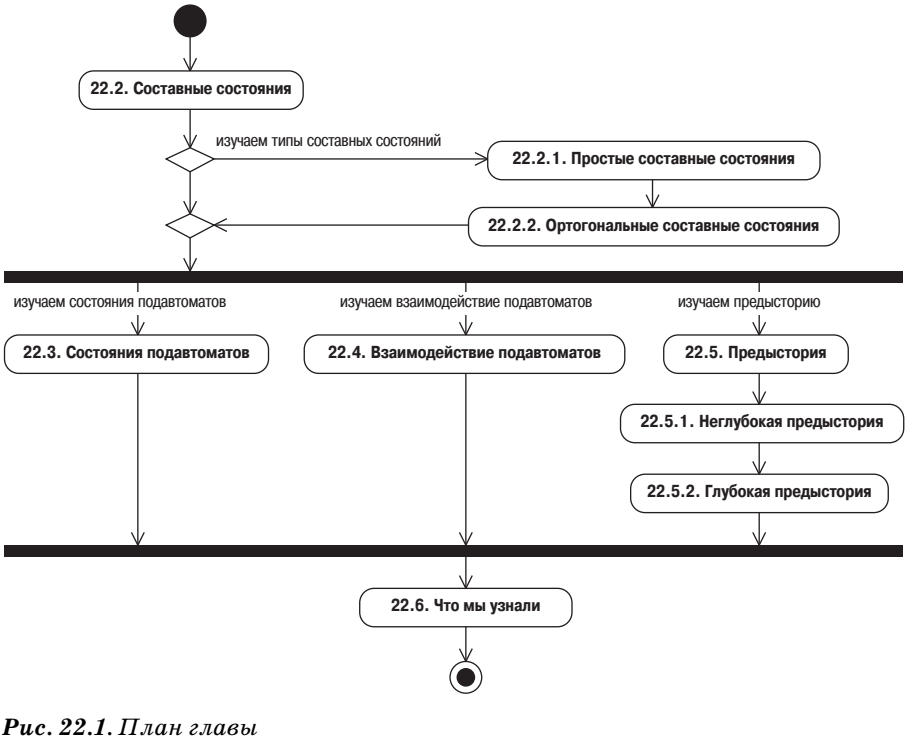


Рис. 22.1. *План главы*

вводится понятие вложенных конечных автоматов, или подавтоматов. Затем обсуждаются два типа составных состояний – простое составное состояние (22.2.1) и ортогональное составное состояние (22.2.2). В разделе 22.3 мы рассматриваем, как можно с помощью состояний подавтоматов показать конечные автоматы, представленные на других диаграммах.

При наличии двух или более параллельных подавтоматов между ними часто необходимо установить некоторое взаимодействие. Это обсуждается в разделе 22.4. Здесь вводится стратегия взаимодействия, в которой используются значения атрибутов контекстного объекта.

В разделе 22.5 вводится понятие предыстории, заключающееся в запоминании суперсостоянием своего последнего подсостояния перед исходящим переходом. В разделах 22.5.1 и 22.5.2 обсуждаются два вида предыстории – неглубокая и глубокая.

22.2. Составные состояния

> Составные состояния содержат один или более вложенных подавтоматов.

Составное состояние – это состояние, содержащее вложенные состояния.

Эти вложенные состояния объединяются в один или более конечных автоматов, которые называют *подавтоматами* (*submachines*). Для каждого подавтомата в пиктограмме композиции отведена собственная *область*. Области – это просто участки пиктограммы состояния, разделенные пунктирными линиями. Простой пример областей представлен на рис. 22.2.

> Вложенные подсостояния наследуют все переходы содержащих их суперсостояний.

Вложенные состояния наследуют все переходы состояний, в которые они входят. Это очень важный момент. Если, например, у самого составного состояния есть определенный переход, каждое из вложенных в него состояний тоже имеет этот переход.

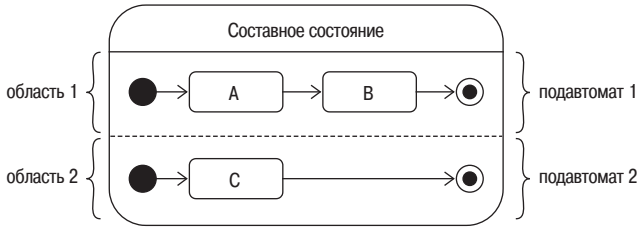


Рис. 22.2. *Каждому подавтомату отведена своя область*

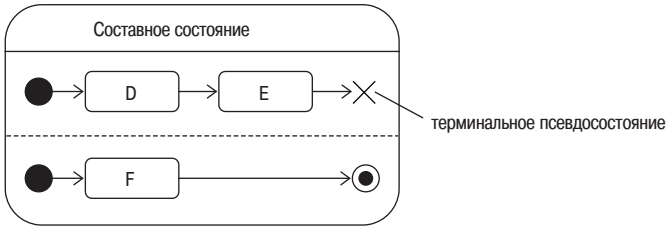


Рис. 22.3. *Терминальное псевдосостояние*

Конечное псевдосостояние подавтомата *относится только к его области*. Таким образом, если подавтомат области 1 в примере на рис. 22.2 достигает своего конечного состояния первым, эта область завершается, а область 2 продолжает выполнение. Если необходимо завершить выполнение всего составного состояния, можно использовать терминальное (terminate, завершающее) псевдосостояние, как показано на рис. 22.3. В этом примере все составное состояние завершается при достижении терминального псевдосостояния.

Вложенные состояния также могут быть составными состояниями. Однако, как правило, необходимо по возможности стремиться к тому, чтобы вложенность составных состояний не превышала двух или трех уровней. При большей глубине вложенности автомат сложно воспринимать на диаграмме и понимать.

Для сохранения ясности и простоты диаграммы состояний иногда необходимо скрывать детали составного состояния. Показать, что состояние является составным, можно без явного отображения его структуры, добавив *пиктограмму композиции* (*composition icon*). Это необязательное дополнение, но очень полезное для обозначения того, что у состояния есть составные части, поэтому мы рекомендуем постоянно им пользоваться. Пиктограмма композиции приведена на рис. 22.4.

В зависимости от количества областей выделяют два типа составных состояний.

1. Простое составное состояние – только одна область.
2. Ортогональное составное состояние – две или более областей.

Эти типы составных состояний рассматриваются в двух следующих подразделах.

Рис. 22.4. *Пиктограмма композиции указывает, что состояние составное*

22.2.1. Простые составные состояния

> Простые составные состояния содержат всего один вложенный конечный автомат.

Простое составное состояние – это составное состояние, содержащее всего одну область. Например, на рис. 22.5 показан автомат для класса ISPDialer (модем). Этот класс отвечает за подключение к провайдеру Интернета. Состояние DialingISP – простое составное состояние, потому что оно имеет всего одну область. В этом конечном автомате есть несколько интересных моментов.

- Переход из DialingISP инициируется событием cancel, наследуемым всеми подсостояниями подавтомата DialingISP. Это *очень* удобно, поскольку означает, что *всегда* при получении события cancel будет осуществляться переход из любого подсостояния в состояние NotConnected (не соединен). Такое использование суперсостояний и подсостояний может существенно упростить автомат.
- У подавтомата DialingISP одно входное псевдосостояние dial (набор номера) и два выходных псевдосостояния, notConnected и connected. Входное псевдосостояние изображается в виде кружка, который обычно размещается на рамке составного состояния (хотя также может находиться внутри нее), и обозначает точку входа в подавтомат. Аналогично выходное псевдосостояние, которое изображается в виде кружка с перекрестьем, обозначает выход из подавтомата.

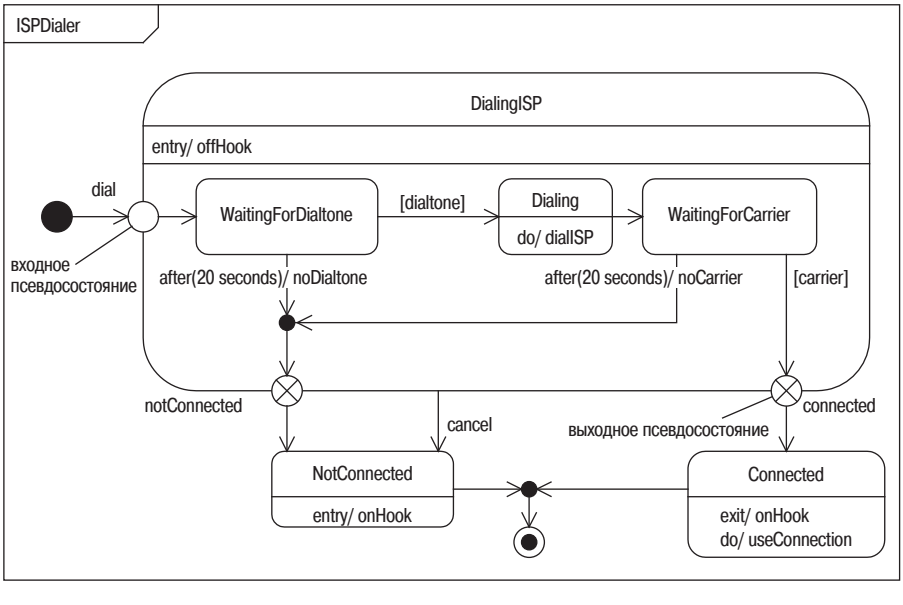


Рис. 22.5. *Конечный автомат класса ISPDialer*

Входные и выходные псевдосостояния очень полезны, поскольку позволяют определять разные пути входа и выхода из составных состояний. Они обеспечивают точки подключения, к которым подключаются переходы в/из других состояний.

Вот полный пошаговый анализ автомата класса ISPDialer.

1. Вход в суперсостояние DialingISP осуществляется через входное псевдосостояние dial, и сразу же выполняется входное действие offHook (поднятие трубки) – модем «поднимает трубку».
2. Осуществляется вход в единственную область и состояние WaitingForDialtone (ожидание тонального сигнала готовности).
3. Ожидание в состоянии WaitingForDialtone длится максимум 20 секунд.
4. Если тональный сигнал готовности не поступает в течение 20 секунд:
   4.1. осуществляется действие noDialtone (нет тонального сигнала готовности) и переход через выходное псевдосостояние notConnected в состояние NotConnected;
   4.2. при входе в NotConnected телефон возвращается на рычаг (действие onHook);
   4.3. осуществляется переход в состояние stop (остановка).
5. Если тональный сигнал готовности получен (т. е. сторожевое условие [dialtone] принимает значение true) в течение 20 секунд:
   5.1. переходим в состояние Dialing (набор), в котором осуществляется деятельность dialISP;
   5.2. по завершении деятельности dialISP происходит автоматический переход в состояние WaitingForCarrier (ожидание несущей);
   5.3. ожидание в состоянии WaitingForCarrier длится максимум 20 секунд.
   5.4. Если в течение 20 секунд несущая не получена:
      5.4.1. осуществляется действие noCarrier (нет несущей) и переход через выходное псевдосостояние notConnected в состояние NotConnected;
      5.4.2. при входе в состояние NotConnected телефон возвращается на рычаг;
      5.4.3. осуществляется переход в состояние stop.
   5.5. Если в течение 20 секунд несущая получена:
      5.5.1. происходит автоматический переход из суперсостояния DialingISP через выходное псевдосостояние connected в состояние Connected;
      5.5.2. осуществляется действие useConnection (использование соединения) до его завершения;
      5.5.3. при выходе из Connected телефон возвращается на рычаг;

5.5.4. осуществляется переход в состояние stop.

6. Если в *любой момент* нахождения в суперсостоянии DialingISP поступает событие cancel:
   6.1. немедленно осуществляется переход в состояние NotConnected;
   6.2. при входе в состояние NotConnected телефон возвращается на рычаг;
   6.3. осуществляется переход в состояние stop.

22.2.2. Ортогональные составные состояния

> Ортогональные составные состояния содержат два или более вложенных подавтоматов, выполняющихся параллельно.

Ортогональные составные состояния содержат два или более выполняющихся параллельно подавтоматов.

При входе в составное состояние начинается одновременное выполнение всех его подавтоматов – это неявное ветвление.

Существует два способа выхода из составного состояния.

1. Оба подавтомата завершаются – неявное объединение.
2. Один из подавтоматов переходит в состояние, находящееся *вне* суперсостояния, обычно через выходное псевдосостояние. Это *не* приводит к объединению – не происходит синхронизации подавтоматов, и остальные подавтоматы просто прерывают выполнение.

Для исследования параллельных составных состояний нужна система, предоставляющая некоторую степень параллелизма. Мы моделируем простую систему сигнализации, которая состоит из блока управления, датчика безопасности, пожарного датчика и блока сигнализации. Автомат всей системы представлен на рис. 22.6.

Этот автомат отражает две основные характеристики системы сигнализации.

1. Если срабатывает датчик вторжения, блок сигнализации воспроизводит тревогу вторжения в течение 15 минут, после чего система возвращается в исходное состояние Monitoring. Это регулируется местным законодательством.
2. Если в ходе воспроизведения тревоги вторжения возникает пожар, происходит немедленный переход из состояния SoundingIntruderAlarm в состояние SoundingFireAlarm и начинает звучать сигнал пожарной тревоги. Это означает, что пожарная тревога всегда имеет более высокий приоритет, чем тревога вторжения.

Initializing и Monitoring – составные состояния. Развернутый вид составного состояния Initializing представлен на рис. 22.7.

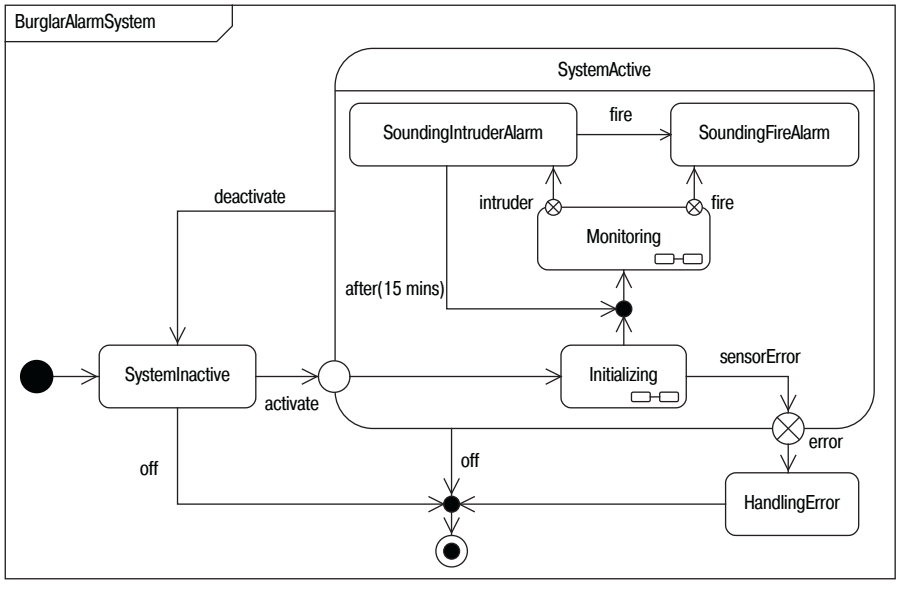


Рис. 22.6. *Автомат системы сигнализации*

При входе в это состояние происходит ветвление и начинается параллельное выполнение двух подавтоматов. В верхнем подавтомате состояние InitializingFireSensors выполняет процесс инициализации пожарных датчиков. В нижнем подавтомате состояние InitializingSecuritySensors делает то же самое для датчиков безопасности.

В нормальных условиях по завершении *обоих* подавтоматов происходит автоматический выход из суперсостояния Initializing. Это объединение: подавтоматы синхронизируются таким образом, что дальнейшая работа невозможна, пока не будут инициализированы *и* пожарные датчики, *и* датчики безопасности. Конечно, продолжительность инициализации

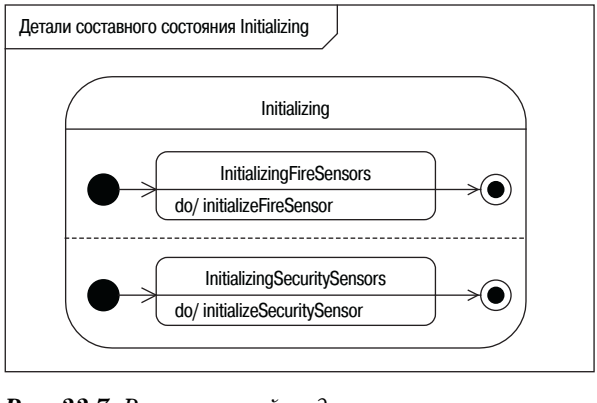


Рис. 22.7. *Развернутый вид составного состояния Initializing*

зависит от используемых типов датчиков, но в простейшем случае это может быть просто короткая задержка на «разогрев» приборов.

Из состояния Initializing также есть переход sensorError (ошибка датчика) (рис. 22.6), наследуемый обоими подсостояниями. Он позволяет немедленно выйти из Initializing при возникновении ошибки датчика. Также составное состояние Initializing и все его подсостояния наследуют переход off от своего суперсостояния SystemActive (система активна). Он обеспечивает возможность немедленного выхода из Initializing (и всех подсостояний SystemActive) при получении события off.

Иногда требуется запустить параллельные потоки управления, которые *не* надо синхронизировать с помощью объединения по их завершении. Этот вариант представлен составным состоянием Monitoring, показанным на диаграмме состояний на рис. 22.8. У этого состояния есть некоторые интересные свойства.

- Два подавтомата *не* синхронизируются:
  - При возникновении события fire осуществляется явный переход от MonitoringFireSensors в выходное псевдосостояние fire, при этом происходит выход из состояния Monitoring. Подавтомат MonitoringFireSensors завершается, но подавтомат MonitoringSecuritySensors продолжает выполнение.
  - Аналогично при возникновении события intruder осуществляется явный переход от MonitoringSecuritySensors в выходное псевдосостояние intruder, при этом происходит выход из суперсостояния Monitoring. Подавтомат MonitoringSecuritySensors завершается, но подавтомат MonitoringFireSensors продолжает выполнение.
- Составное состояние Monitoring и все его подсостояния (рис. 22.8) наследуют переход off от своего суперсостояния SystemActive, представленного на рис. 22.6. Это обеспечивает системе возможность немедленно завершить работу в ответ на событие off независимо от того, какое из подсостояний является активным.

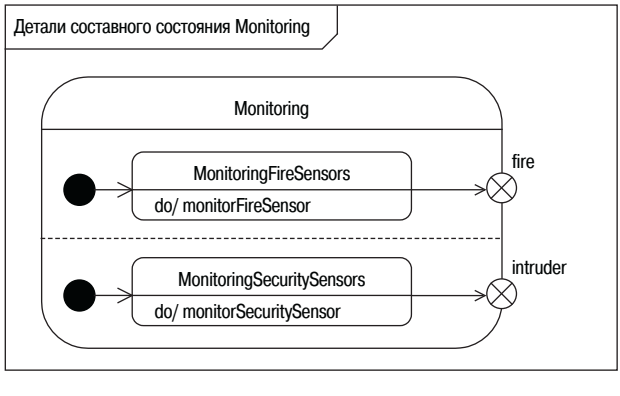


Рис. 22.8. *Составное состояние Monitoring*

Этот пример показывает, как использование параллельных составных состояний, с синхронизацией или без нее, позволяет очень эффективно моделировать параллелизм.

22.3. Состояния подавтоматов

> Состояние подавтомата ссылается на другой автомат.

Состояние подавтомата – это особое состояние, ссылающееся на конечный автомат, представленный на отдельной диаграмме. Это немного похоже на вызов конечным автоматом подпрограммы другого конечного автомата. Состояния подавтоматов семантически эквивалентны составным состояниям.

Состояния подавтоматов могут использоваться для упрощения сложных автоматов. Конечные автоматы разделяются на отдельные диаграммы, и затем основная диаграмма ссылается на них с помощью состояний подавтоматов.

Состояния подавтоматов могут предоставить способ повторного использования поведения. Поведение описывается на одной диаграмме, и затем при необходимости просто делается ссылка на эту диаграмму. Например, пусть две очень похожие системы охранной сигнализации имеют некоторое сходное поведение. Это поведение можно описать на отдельной диаграмме, а затем в автоматах каждой системы ссылаться на эту диаграмму с помощью состояний подавтоматов.

Состояния подавтоматов именуются следующим образом:

```
имя состояния : имя используемой диаграммы состояний
```

На рис. 22.9 представлена диаграмма состояний, описывающая поведение, которое используется в другой диаграмме. Обратите внимание, что входное и выходное псевдосостояния можно показать *на* рамке.

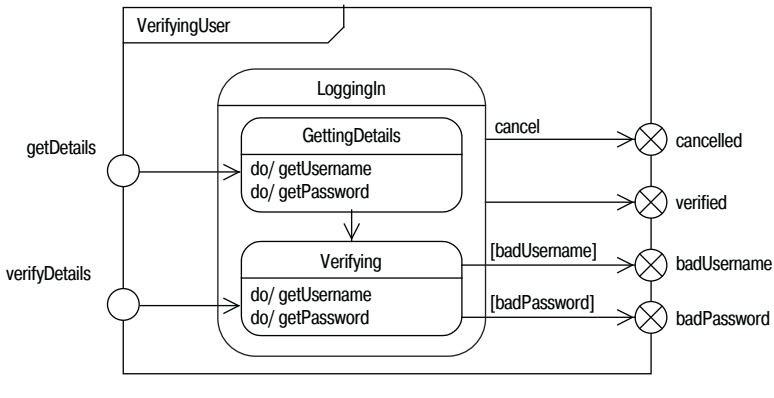


Рис. 22.9. *Диаграмма состояний автомата VerifyingUser*

Их также можно поместить внутрь рамки, но нам кажется, что размещение *на* рамке более четко отображает их суть как точек входа и выхода конечного автомата.

Сослаться на автомат VerifyingUser (верификация пользователя) можно с помощью состояния подавтомата, как показано на рис. 22.10. Состояние подавтомата названо verifyingCustomer (верификация клиента). Читая эту диаграмму, состояние подавтомата необходимо мысленно заменять содержимым диаграммы VerifyingUser.

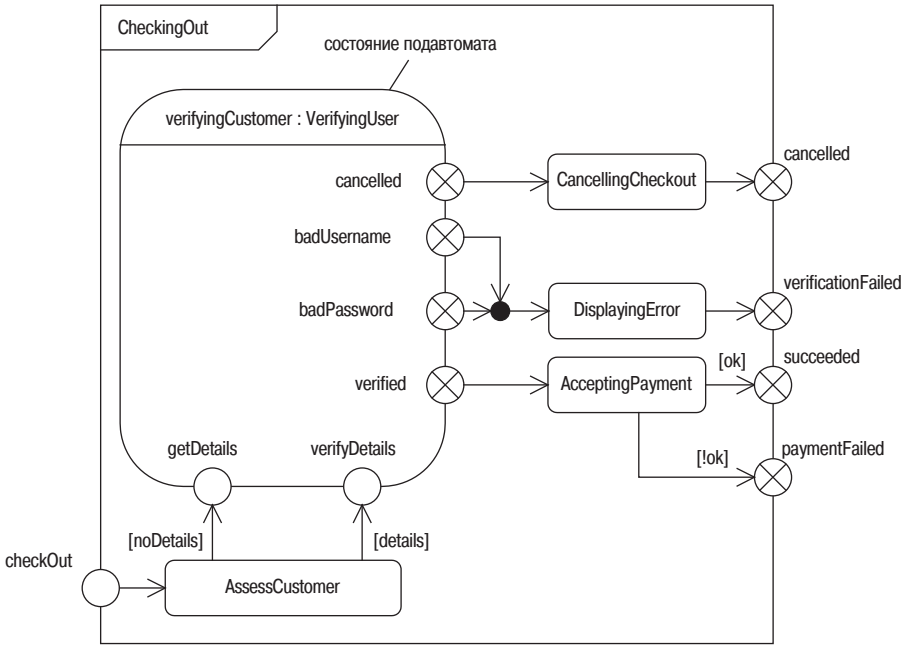


Рис. 22.10. *Ссылка на автомат VerifyingUser с помощью состояния подавтомата*

22.4. Взаимодействие подавтоматов

На рис. 22.7 было показано, как можно использовать ветвления и объединения для создания и последующей синхронизации параллельных подавтоматов. Это вид синхронного взаимодействия подавтоматов – параллельные подавтоматы ожидают завершения друг друга, пока *все* они не закончат выполнение.

Однако очень часто необходимо обеспечить взаимодействие подавтоматов, но без синхронизации автоматов. Это называют асинхронным взаимодействием.

В UML асинхронное взаимодействие может быть обеспечено с помощью «сообщений», или «флагов», оставляемых одним подавтоматом во время выполнения. Другие подавтоматы могут подбирать эти флаги, когда будут готовы.

> Для асинхронного взаимодействия параллельных подавтоматов могут использоваться значения атрибутов.

Такие флаги создаются с помощью значений атрибутов контекстного объекта автомата. Основная стратегия состоит в том, что один подавтомат задает значения атрибутов, а другие подавтоматы используют их в сторожевых условиях своих переходов.

В состоянии OrderProcessing, показанном на рис. 22.11, нельзя предсказать, что произойдет раньше: комплектация заказа или его оплата. Для комплектации некоторых заказов, возможно, понадобится ждать поступления новых деталей, а некоторые могут быть взяты прямо со склада. Аналогично некоторые платежи могут быть более или менее срочными (по кредитной карте, например), а для прохождения других может потребоваться несколько рабочих дней (по чеку, например). Однако здесь действует бизнес-правило, выстраивающее логическую зависимость между двумя подавтоматами: заказ не может быть доставлен, пока не будет укомплектован *и* оплачен.

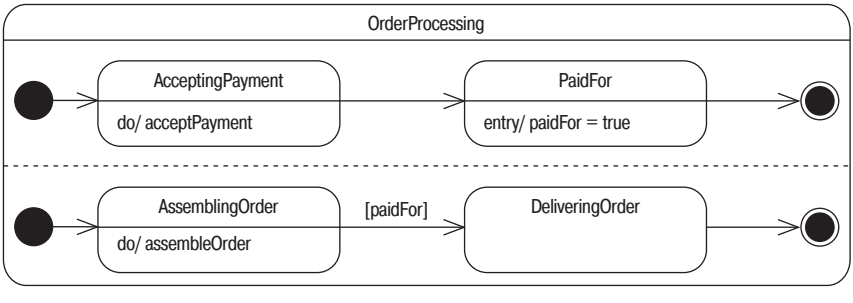


Рис. 22.11. *Параллельное выполнение подавтоматов*

В верхнем подавтомате на рис. 22.11 по завершении состояния AcceptingPayment осуществляется переход в состояние PaidFor, где атрибуту paidFor присваивается значение true. В нижнем подавтомате, когда завершается AssemblingOrder, можно осуществить переход к DeliveringOrder, но *только* когда атрибут paidFor принимает значение true. Мы добились асинхронного взаимодействия двух подавтоматов с помощью атрибута, выступающего в роли флага, значение которого задается одним подавтоматом, а запрашивается другим. Это простой и широко используемый механизм. И наконец, оба подавтомата завершаются и синхронизируются, и мы покидаем состояние OrderProcessing.

22.5. Предыстория

При использовании автоматов в моделировании часто сталкиваешься со следующей ситуацией.

- Вы находитесь в подсостоянии A составного состояния.

- Переходите из составного состояния (и, следовательно, из подсостояния A).
- Проходите через одно или более внешних состояний.
- Возвращаетесь в составное состояние, *но* хотели бы продолжить выполнение в подсостоянии A с того момента, на котором остановились в прошлый раз.

Как можно добиться этого? Понятно, что составному состоянию необходим какой-то способ для запоминания подсостояния, в котором вы находились при выходе из него. Это требование возвращения к тому, на чем остановились, настолько распространено, что специально для его реализации в UML было включено псевдосостояние предыстории.

> Предыстория позволяет суперсостоянию при возвращении после прерывания «начинать с того, на чем остановился».

С помощью предыстории суперсостояния запоминают последнее активное подсостояние. Существует два типа псевдосостояний предыстории – неглубокая и глубокая предыстории. Рассмотрим их по очереди.

22.5.1. Неглубокая предыстория

На рис. 22.12 показан конечный автомат прецедента BrowseCatalog системы электронной коммерции.

В данном примере из суперсостояния Browsing можно выйти по трем событиям:

- exit – завершает автомат и возвращается на то место, где осуществлялось выполнение до него (нет необходимости рассматривать этот случай более подробно);
- goToBasket – осуществляет переход в составное состояние DisplayingBasket, в котором отображается текущее содержимое корзины для покупок;
- goToCheckout – осуществляет переход в составное состояние CheckingOut, в котором покупателю представляется заказ с перечислением всех покупок.

При возвращении в состояние Browsing из DisplayingBasket или CheckingOut было бы неплохо, чтобы пользователи попадали именно туда, где остановились. Только такое поведение имеет смысл.

> Неглубокая предыстория запоминает последнее подсостояние того же уровня, что и псевдосостояние неглубокой предыстории.

Псевдосостояние неглубокой предыстории может иметь множество входящих переходов, но только один исходящий. Псевдосостояние неглубокой предыстории *запоминает*, в каком подсостоянии вы находились при выходе из суперсостояния. Если затем происходит возвращение из

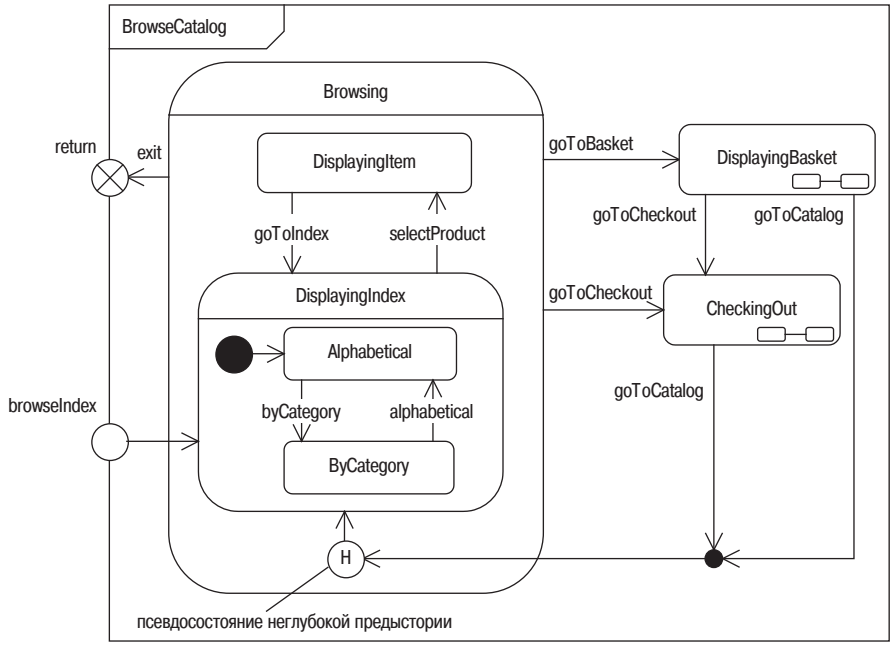


Рис. 22.12. *Неглубокая предыстория*

внешнего состояния в состояние предыстории, индикатор автоматически перенаправляет переход в последнее подсостояние, которое было запомнено (в данном случае DisplayingIndex или DisplayingItem). Если вход в суперсостояние осуществляется впервые, такого подсостояния *нет*. Поэтому срабатывает единственный исходящий переход индикатора состояния предыстории, и вы переходите в состояние DisplayingIndex.

Благодаря предыстории суперсостояния запоминают последнее подсостояние, которое было активно при выходе из суперсостояния. Неглубокая предыстория обеспечивает запоминание подсостояния, находящегося *на том же уровне*, что и сам индикатор состояния предыстории. Однако на рис. 22.12 можно увидеть, что DisplayingIndex само является составным состоянием. Неглубокая предыстория не обеспечивает запоминание подсостояний этого состояния. Для этого необходима глубокая предыстория.

22.5.2. Глубокая предыстория

> Глубокая предыстория запоминает последнее подсостояние того же или более низкого уровня, чем псевдосостояние глубокой предыстории.

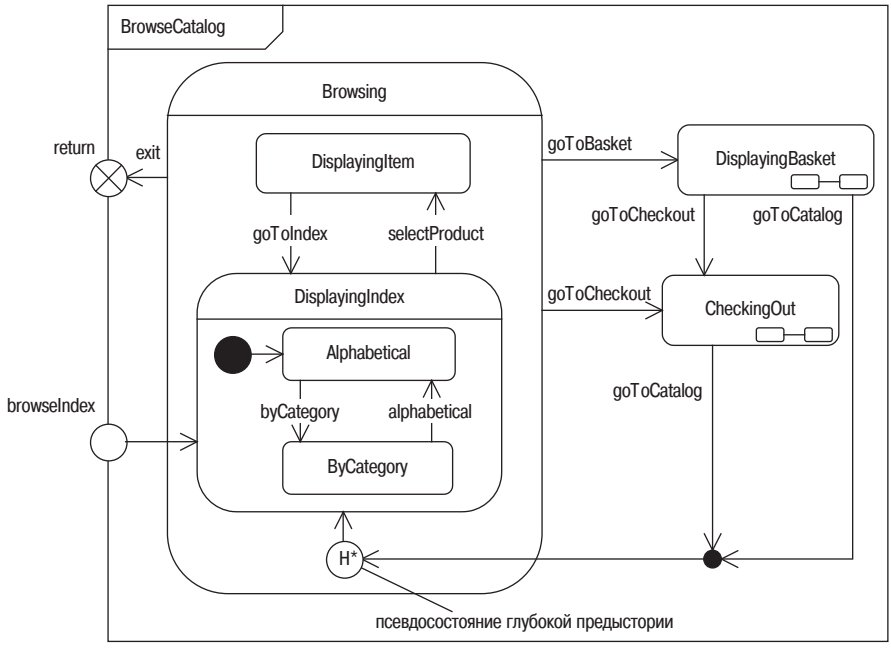


Рис. 22.13. *Глубокая предыстория*

С помощью глубокой предыстории можно запоминать не только активное подсостояние того же уровня, что и псевдосостояние предыстории, но и подподсостояния любого уровня вложенности.

На рис. 22.13 автомат был модифицирован для использования глубокой предыстории. В данном случае мы будем не только возвращаться в состояние DisplayingIndex или DisplayingItem, но и восстанавливать тип индекса (Alphabetical или ByCategory). Это можно было бы смоделировать и *без* применения глубокой предыстории, но все было бы намного сложнее.

Как и неглубокая, глубокая предыстория может иметь множество входящих переходов, но только один исходящий переход. Исходящий переход срабатывает при первом входе в суперсостояние, когда *нет* последнего подсостояния, которое было запомнено.

22.6. Что мы узнали

UML предоставляет богатый синтаксис конечных автоматов, который позволяет представлять сложное поведение в виде лаконичных автоматов. Мы узнали следующее:

- Составные состояния могут содержать один или более вложенных подавтоматов – подсостояния наследуют все переходы своего суперсостояния.

    - Каждый подавтомат существует в собственной области.
    - Конечное псевдосостояние действует только в рамках области.
    - Терминальное псевдосостояние используется для завершения всех областей.
- Последовательное составное состояние содержит только один вложенный подавтомат.
- Параллельное составное состояние содержит два или более вложенных подавтоматов, которые выполняются параллельно.
    - При входе в состояние происходит ветвление, и подавтоматы начинают свое параллельное выполнение.
    - Если у всех подавтоматов есть состояние остановки, суперсостояние нельзя покинуть, пока не будут завершены все подавтоматы; это объединение.
    - Если подавтоматы осуществляют явные переходы во внешние состояния, можно покинуть суперсостояние без объединения.
- Состояние подавтомата – это ссылка на другой конечный автомат:
    - упрощает сложные конечные автоматы;
    - обеспечивает повторное использование конечных автоматов.
- Взаимодействие подавтоматов:
    - значения атрибутов – один подавтомат задает значение атрибута, а другие подавтоматы проверяют это значение.
- Предыстория позволяет суперсостоянию запоминать последнее подсостояние перед исходящим переходом.
    - Неглубокая предыстория позволяет суперсостоянию перед исходящим переходом запоминать последнее подсостояние *того же* уровня, что и само псевдосостояние неглубокой предыстории:
        - при возвращении в псевдосостояние неглубокой предыстории переход направляется в последнее подсостояние, которое было запомнено;
        - при первом входе (последнее подсостояние отсутствует) срабатывает единственный выходной переход псевдосостояния неглубокой предыстории.
    - Глубокая предыстория позволяет суперсостоянию перед исходящим переходом запоминать последнее подсостояние *любого* уровня:
        - при возвращении в псевдосостояние глубокой предыстории переход направляется в последнее подсостояние, которое было запомнено;
        - при первом входе (последнее подсостояние отсутствует) срабатывает единственный выходной переход псевдосостояния глубокой предыстории.

V

Реализация

23

Рабочий поток реализации

23.1. План главы

В рабочем потоке реализации для ОО аналитика или проектировщика работы очень мало, поэтому эта часть книги самая маленькая. Тем не менее здесь есть некоторые вопросы, требующие внимательного рассмотрения. Хотя основная деятельность потока реализации – производство кода, здесь по-прежнему присутствуют некоторые элементы UML-моделирования.

23.2. Рабочий поток реализации

Рабочий поток реализации всерьез начинается в фазе Уточнение и является основным потоком фазы Построение (рис. 23.2).

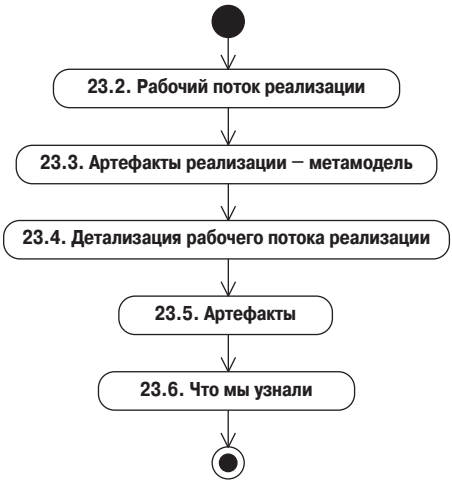


Рис. 23.1. *План главы*

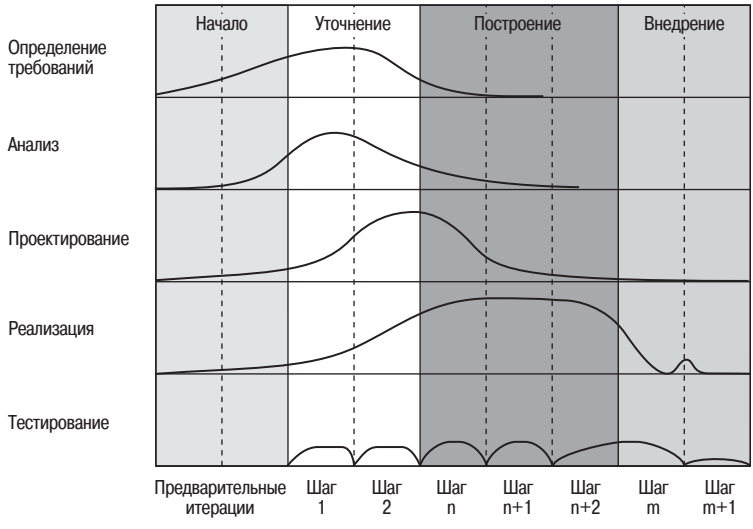


Рис. 23.2. *Рабочий поток реализации. Адаптировано с рис. 1.5 [Jacobson 1] с разрешения издательства Addison-Wesley*

> Поток реализации – основной поток фазы Построение.

Реализация состоит в преобразовании проектной модели в исполняемый код. С точки зрения аналитика или проектировщика цель реализации – производство модели реализации, если в этом возникает необходимость. Эта модель включает распределение (преимущественно тактическое) проектных классов по компонентам. Как это делается, в большой степени зависит от целевого языка программирования.

> Реализация состоит в преобразовании проектной модели в исполняемый код.

Основное внимание в процессе реализации направлено на производство исполняемого кода. Создание модели реализации может быть побочным продуктом этого процесса, но не явной деятельностью моделирования. На самом деле многие инструментальные средства моделирования позволяют создавать модель реализации из исходного кода путем обратного проектирования. Это предоставляет программистам возможность эффективно выполнять моделирование реализации.

Однако есть два случая, когда очень важно, чтобы опытные аналитики или проектировщики провели явное моделирование реализации.

- Если предполагается генерировать код прямо из модели, понадобится определить такие детали, как исходные файлы и компоненты (если не используются применяемые по умолчанию значения у средства моделирования).

- Если осуществляется компонентно-ориентированная разработка (CBD) с целью повторного использования компонентов, распределение проектных классов и интерфейсов по компонентам становится стратегически важным вопросом. Вероятно, вы захотите сначала смоделировать эту часть проекта, а не перекладывать все на плечи одного программиста.

В этой главе рассматривается, что включает в себя процесс создания модели реализации.

23.3. Артефакты реализации – метамодель

Отношения между моделью реализации и проектной моделью очень просты. Фактически модель реализации – это представление проектной модели с точки зрения реализации. Иными словами, модель реализации – это *часть* проектной модели, что и отражено на рис. 23.3.

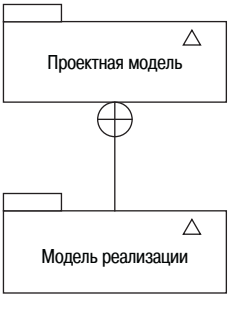


Рис. 23.3. *Модель реализации как часть проектной модели*

> Модель реализации – это часть проектной модели.

Модель реализации – это часть проектной модели, занимающаяся вопросами реализации. Она определяет, как проектные элементы представляются артефактами и как эти артефакты развертываются на узлах. Артефакты представляют описания реальных сущностей, таких как исходные файлы, а узлы представляют описания оборудования или сред выполнения, в которых эти сущности развертываются. Отношения между проектной моделью и моделью реализации показаны на рис. 23.4.

Отношение «manifest» между артефактами и компонентами указывает на то, что артефакты являются физическими представлениями компонентов. Например, компонент может состоять из класса и интерфейса, которые реализованы единственным артефактом: файлом, содержащим исходный код.

Проектные компоненты – это логические сущности, которые группируют проектные элементы. А артефакты реализации проецируются на

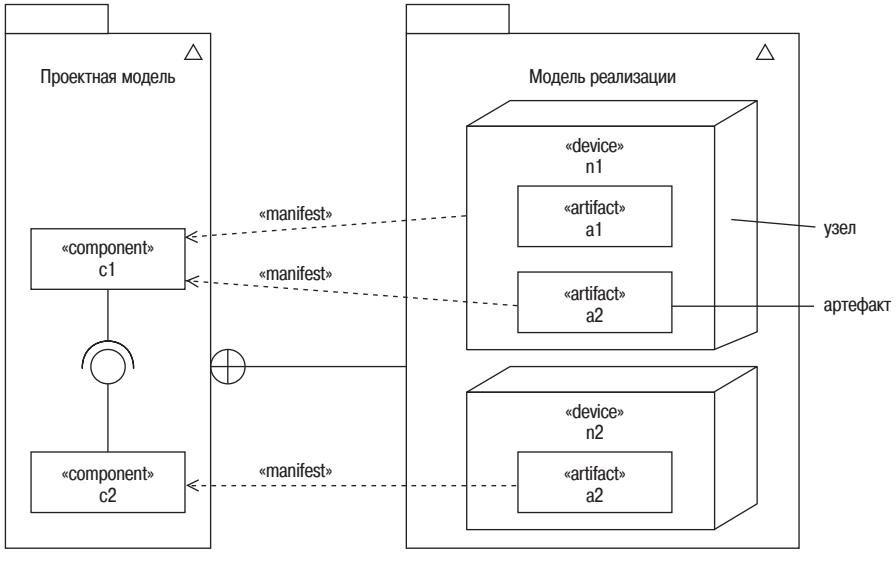


Рис. 23.4. *Отношения между проектной моделью и моделью реализации*

реальные, физические механизмы группировки целевого языка реализации.

23.4. Детализация рабочего потока реализации

Как видно из рис. 23.5, в рабочем потоке реализации участвуют архитектор, системный интегратор и разработчик компонентов. В любой из этих трех ролей могут выступать отдельные аналитики или проектировщики или небольшие команды аналитиков или проектировщиков. Их основное внимание будет направлено на производство моделей развертывания и реализации (часть реализации архитектуры). Интегрирование системы, реализация классов и тестирование элементов (unit testing) выходят за рамки рассмотрения этой книги – это вопросы реализации, а не анализа и проектирования. (Обратите внимание, что на рис. 23.5 в оригинальный рисунок было внесено изменение: Реализация подсистемы заменена на более общую Реализацию компонента, поскольку это более правильно с точки зрения UML 2.)

23.5. Артефакты

Основной артефакт рабочего потока реализации с точки зрения ОО аналитика или проектировщика – модель реализации. Эта модель состоит из диаграмм компонентов, показывающих, как артефакты представляют компоненты, и диаграммы нового типа – диаграммы развертывания. Диаграмма развертывания моделирует физические вычислительные узлы, на которых будут развертываться программ-

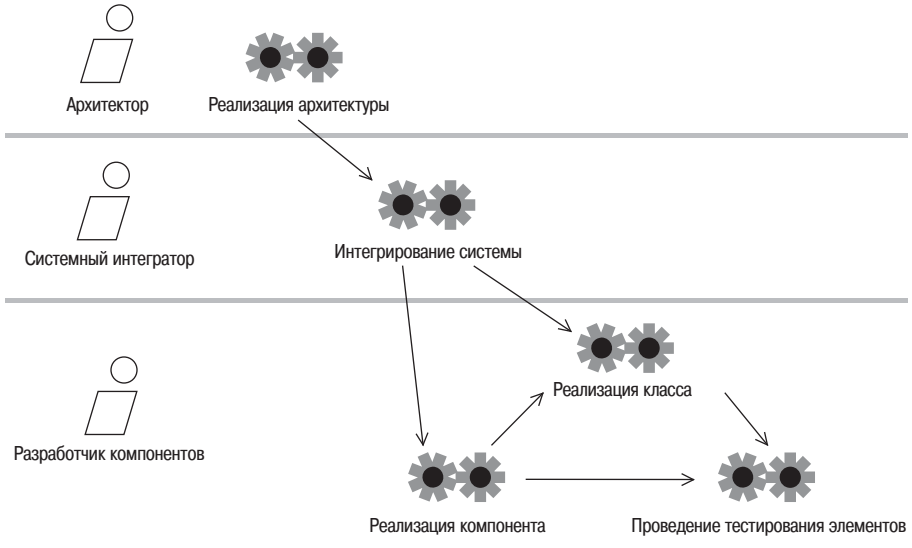


Рис. 23.5. *Рабочий поток реализации. Адаптировано с рис. 10.16 [Jacobson 1] с разрешения издательства Addison-Wesley*

ные артефакты, и отношения между этими узлами. Подробно диаграммы развертывания рассматриваются в главе 24.

23.6. Что мы узнали

Реализация главным образом касается создания кода. Однако ОО аналитик или проектировщик может быть привлечен для создания модели реализации. Мы узнали следующее:

- Рабочий поток реализации – основной поток фазы Построения.
- Реализация заключается в преобразовании проектной модели в исполняемый код.
- Моделирование реализации имеет важное значение, если:
  - предполагается использовать модель при прямой разработке (генерации кода);
  - осуществляется CBD с целью обеспечения повторного использования кода.
- Модель реализации – часть проектной модели.
- Артефакты представляют описания реальных сущностей, таких как исходные файлы:
  - компоненты представляются артефактами;
  - артефакты развертываются на узлах.
- Узлы представляют описания оборудования и сред выполнения.

24

Развертывание

24.1. План главы

В этой главе рассматриваются деятельность UP Реализация архитектуры (Architectural implementation) и способ создания диаграммы развертывания. Эта диаграмма показывает, как разрабатываемое программное обеспечение будет развертываться на физическом оборудовании и как соединяется это оборудование. В разделе 24.5 предлагается простой пример на языке Java.

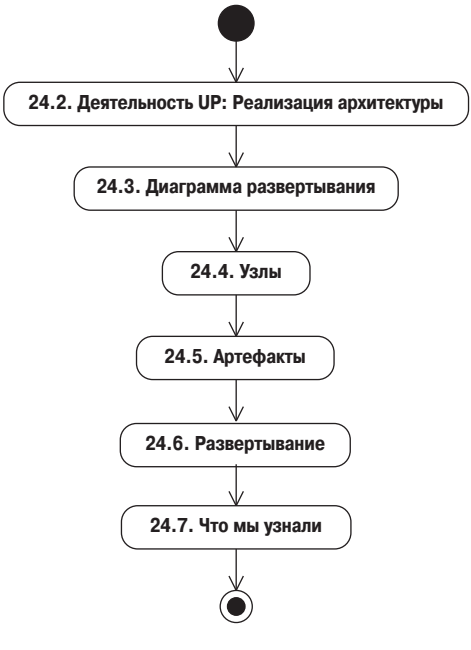


Рис. 24.1. *План главы*

24.2. Деятельность UP: Реализация архитектуры

> Реализация архитектуры состоит в определении важных с точки зрения архитектуры компонентов и проецировании их на физическое оборудование.

Эта деятельность состоит в определении важных с точки зрения архитектуры компонентов и проецировании их на физическое оборудование. То есть здесь мы занимаемся моделированием физической структуры и распределения системы. Ключевая фраза – «важный с точки зрения архитектуры». В принципе можно было бы полностью смоделировать физическое развертывание системы. Но это имело бы небольшую практическую ценность, потому что подробности развертывания многих компонентов с точки зрения архитектуры не очень важны. Исключением является случай, когда код генерируется из модели. Тогда, вероятно, понадобится более подробная модель развертывания, чтобы генератор кода знал, куда помещать выходные артефакты, и мог создавать соответствующие дескрипторы развертывания и компоновать файлы.

Деятельность UP Реализация архитектуры представлена на рис. 24.2. В оригинальный рисунок внесены два изменения (измененные артефакты затушеваны):

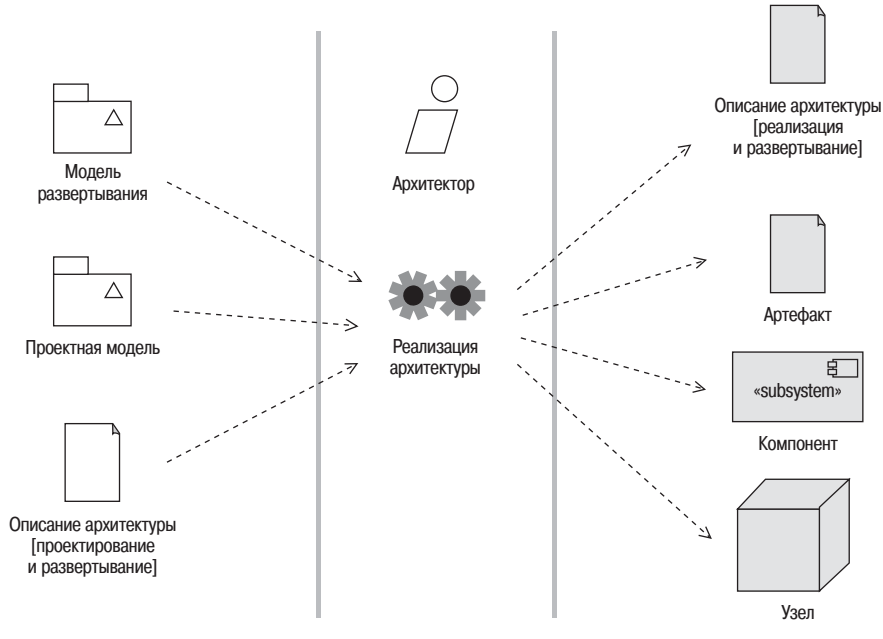


Рис. 24.2. *Деятельность UP Реализация архитектуры. Адаптировано с рис. 10.17 [Jacobson 1] с разрешения издательства Addison-Wesley*

- Согласно UML 2 подсистема показана как обозначенный стереотипом компонент, а не как пакет, обозначенный стереотипом.
- Результирующие артефакты и узлы деятельности показаны явно (в оригинальном рисунке они были неявными).

С точки зрения ОО аналитика/проектировщика основная деятельность Реализации архитектуры – создание одной или более диаграмм развертывания. Диаграмма развертывания объединяет компоненты, артефакты и узлы для определения физической архитектуры системы. Далее глава посвящена подробному обсуждению диаграмм этого типа.

Другая деятельность – дополнение описания архитектуры важными с точки зрения архитектуры деталями развертывания и реализации.

24.3. Диаграмма развертывания

В UML развертывание – это процесс распределения артефактов по узлам или экземпляров артефактов по экземплярам узлов. Скоро мы перейдем к подробному обсуждению артефактов и узлов.

> Диаграмма развертывания проецирует программную архитектуру на аппаратную архитектуру.

Диаграмма развертывания определяет физическое оборудование, на котором будет выполняться программная система, а также описывает, как программное обеспечение развертывается на это оборудование.

Диаграмма развертывания проецирует программную архитектуру, созданную при проектировании, на исполняющую ее физическую архитектуру системы. В распределенных системах она моделирует распределение программного обеспечения по физическим узлам.

Существует две формы диаграмм развертывания.

> Дескрипторная форма диаграммы развертывания – артефакты, развернутые на узлах.

1. Дескрипторная форма (descriptor form) – содержит узлы, отношения между узлами и артефакты. Узел представляет тип оборудования (например, ПК). Аналогично артефакт представляет тип физического программного артефакта, например Java JAR-файл.
2. Экземплярная форма (instance form) – включает экземпляры узлов, отношения между экземплярами узлов и экземпляры артефактов. Экземпляры узлов представляют конкретную, идентифицируемую часть оборудования (например, ПК Джима). Экземпляр артефакта представляет конкретный экземпляр типа программного обеспечения, например определенную копию FrameMaker (*www. adobe.com*), использованную для написания этой книги, или конкретный JAR-

файл. Если детали конкретных экземпляров неизвестны (или неважны), могут использоваться анонимные экземпляры.

> Экземплярная форма диаграммы развертывания – экземпляры артефактов развертываются на экземплярах узлов.

Хотя диаграмма развертывания рассматривается как деятельность рабочего потока реализации, ее первое приближение часто создается при проектировании как часть процесса выбора окончательной аппаратной архитектуры. Можно начать с создания дескрипторной формы диаграммы развертывания, ограничившись узлами и их связями, а затем уточнить ее и превратить в одну или более экземплярных форм, представляющих возможные компоновки анонимных экземпляров узлов. Когда станут известны подробности об оборудовании сайта, на котором будет развертываться проект, при необходимости можно создать экземплярную форму диаграммы развертывания, показывающую фактически используемые на этом сайте компьютеры и устройства.

Таким образом, создание диаграммы развертывания – это процесс из двух этапов:

1. В рабочем потоке проектирования основное внимание сосредоточено на узле или экземплярах узла и соединениях.
2. В рабочем потоке реализации – на распределении экземпляров артефактов по экземплярам узлов (экземплярная форма) или артефактов по узлам (дескрипторная форма).

В следующих двух разделах подробно рассматриваются узлы и артефакты.

24.4. Узлы

> Узел представляет тип вычислительного ресурса.

Спецификация UML 2.0 [UML2S] гласит: «Узел представляет тип вычислительного ресурса, на который могут быть развернуты артефакты для выполнения».

Существует два стандартных стереотипа для узлов:

- «device» (устройство) – узел представляет тип физического устройства, например ПК или сервер Fire корпорации Sun.
- «execution environment» (среда выполнения) – узел представляет тип среды выполнения программного обеспечения, например веб-сервер Apache или EJB-контейнер (Enterprise JavaBeans) JBoss.

Узлы могут быть вложены в узлы. Например, дескрипторная форма диаграммы развертывания на рис. 24.3 показывает, что нуль или более WindowsPC, на которых выполняется веб-броузер Firefox, могут быть со-

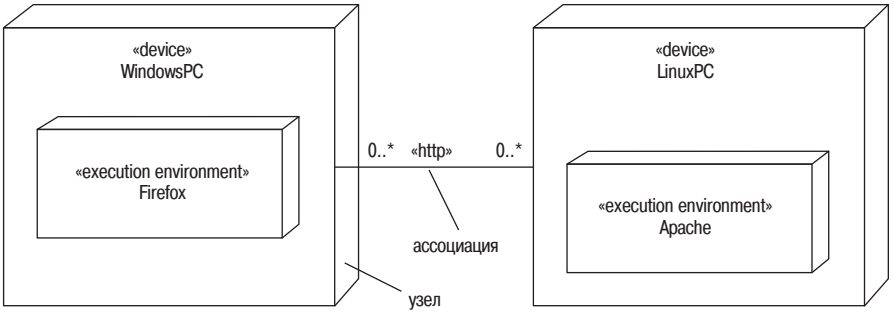


Рис. 24.3. *Дескрипторная форма диаграммы развертывания*

единены с нулем или более веб-серверами Apache, которые выполняются на LinuxPC. Обратите внимание, что назвав узлы WindowsPC и LinuxPC, мы указали *и* тип оборудования (PC), *и* операционную систему, т. е. среду выполнения для всего программного обеспечения, выполняющегося на этих устройствах. Это общепринятая практика, поскольку выделение специального узла среды выполнения для операционной системы загромождает диаграмму. Firefox представлен как среда выполнения, потому что в нем могут выполняться подключаемые (plug-in) компоненты, такие как апплеты Java.

Ассоциация между узлами представляет канал связи, по которому может передаваться информация в обоих направлениях. На рис. 24.3 ассоциация обозначена стереотипом «http». Это указывает на то, что она представляет соединение HTTP (HyperText Transport Protocol[1] – протокол передачи гипертекста) между двумя узлами.

> Экземпляр узла представляет конкретный вычислительный ресурс.

Если необходимо показать конкретные экземпляры узлов, можно использовать экземплярную форму диаграммы развертывания (рис. 24.4). На рисунке изображены два конкретных ПК: компьютеры Джима (JimsPC) и Илы (IlasPC), подключенные к серверу WebServer1, работающему под управлением Linux. В экземплярной форме диаграммы экземпляры представляют реальные физические устройства или реальные экземпляры сред выполнения, выполняющихся на этих устройствах. Имена элементов подчеркнуты, чтобы показать, что они отображают экземпляры узлов.

Дескрипторная форма диаграмм развертывания хороша для моделирования типа физической архитектуры, а экземплярная форма – для моделирования фактического развертывания этой архитектуры на конкретном сайте.

---

[1] или Hypertext Transfer Protocol. – *Примеч. науч. ред.*

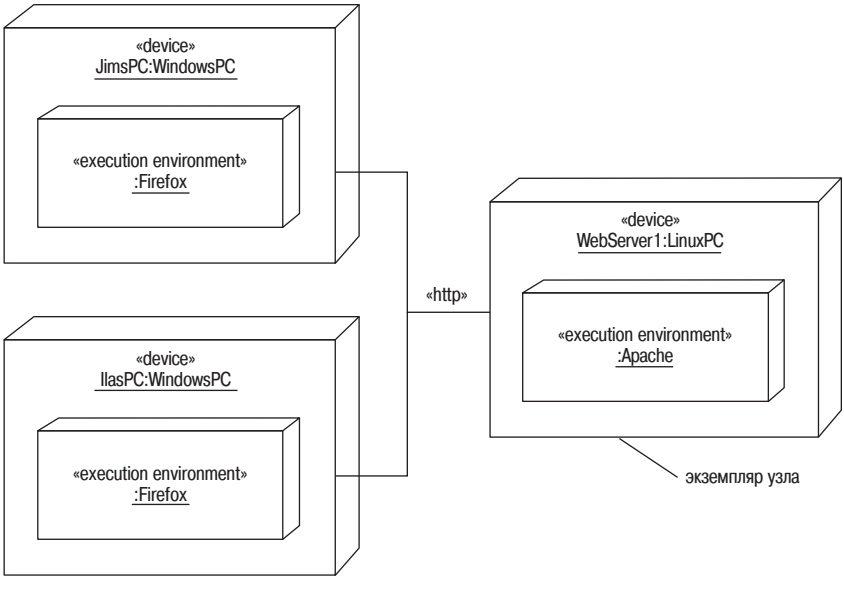


Рис. 24.4. *Экземплярная форма диаграммы развертывания*

Согласно книге «The UML User Guide» [Booch 2], диаграммы развертывания являются самой богатой стереотипами частью UML. Для стереотипов можно придумать собственные пиктограммы, напоминающие реальное оборудование, и затем использовать эти символы на диаграмме развертывания. Такой подход упрощает восприятие диаграммы. Здесь может помочь богатая библиотека изображений! Пример полностью визуализированной дескрипторной формы диаграммы развертывания приведен на рис. 24.5.

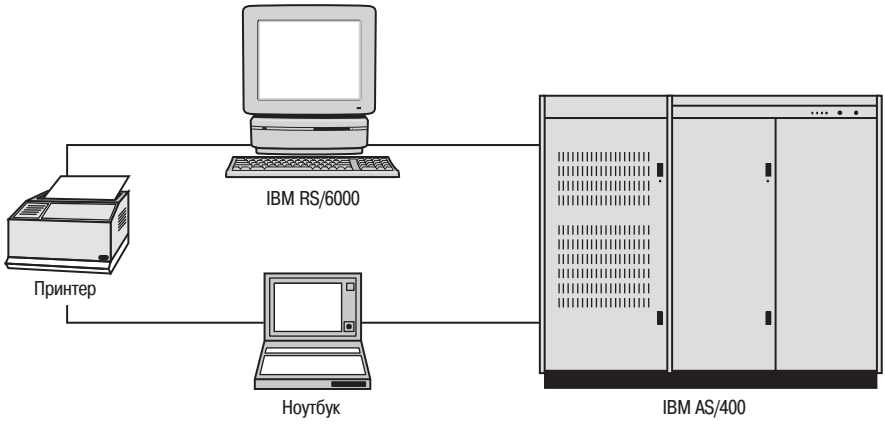


Рис. 24.5. *Визуализированная дескрипторная форма диаграммы развертывания*

24.5. Артефакты

> Артефакт представляет описание реальной сущности, например, такой как файл.

Артефакт представляет описание конкретной, реальной сущности, такой как исходный файл BankAccount.java. Артефакты развертываются на узлах. Некоторые примеры артефактов:

- исходные файлы;
- исполняемые файлы;
- сценарии;
- таблицы баз данных;
- документы;
- результаты процесса разработки, например UML-модель.

> Экземпляр артефакта представляет конкретный *экземпляр* конкретного артефакта.

Экземпляр артефакта представляет определенный экземпляр конкретного артефакта, например определенную копию файла BankAccount.java, развернутую на конкретном компьютере. Экземпляры артефактов развертываются на экземплярах узлов.

> Артефакты могут представлять один или более компонентов.

Артефакты могут обеспечивать физическое представление UML-элементов *любого* типа. Обычно они представляют один или более компонентов, как проиллюстрировано на рис. 24.6. Здесь показан артефакт librarySystem.jar, который представляет три компонента типа «white-box» (учитывающих внутреннюю структуру) – Book, Library и Ticket. Артефакты помечены стереотипом «artifact». В верхнем правом углу артефакта может располагаться пиктограмма артефакта, как показано на рисунке. Здесь также проиллюстрирован тот факт, что артефакты могут зависеть от других артефактов. В данном случае артефакт librarySystem.jar зависит от артефакта jdom.jar.

Наряду с именем в спецификации каждого артефакта есть имя файла (filename), указывающее физическое местоположение артефакта. Например, filename могло бы определять URL, по которому находится оригинал артефакта. Имена файлов экземпляров артефактов указывают на физическое местоположение экземпляра.

Рассмотрим JAR-файл на рис. 24.6 более подробно. Создание этого файла осуществляется в два этапа:

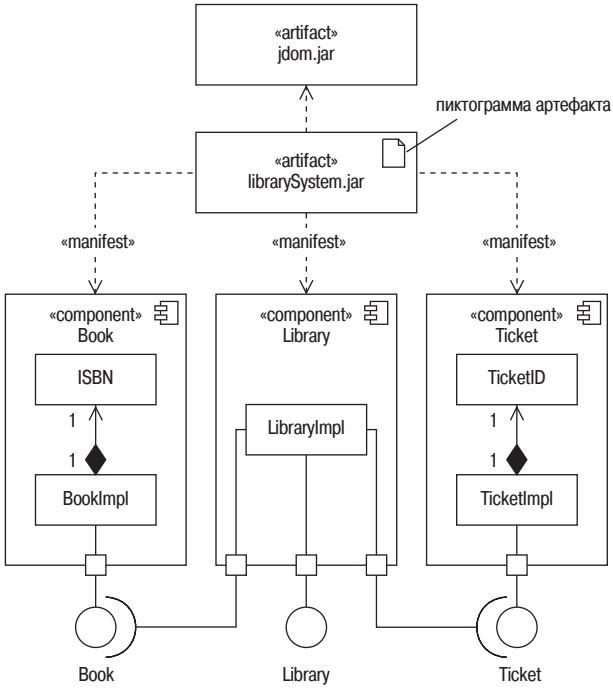


Рис. 24.6. *Артефакт librarySystem.jar представляет три компонента: Book, Library и Ticket*

1. Компилируются исходные Java файлы для классов Book, ISBN, BookImpl, Library, LibraryImpl, Ticket, TicketID и TicketImpl.
2. С помощью Java-инструмента jar[1] из этих откомпилированных файлов создается JAR-файл.

Так создается JAR-файл, изображенный на рис. 24.7. Как видите, он содержит Java-файлы классов для каждого класса и интерфейса системы. В нем также есть каталог META_INF, включающий файл MANIFEST.MF. Этот файл генерируется при помощи jar и описывает содержимое JAR. Рисунок 24.7 также иллюстрирует, как можно показать зависимости между артефактами и артефактами, вложенными в артефакты.

Хотя с точки зрения UML-моделирования рис. 24.7 абсолютно верен, он не отличается особой наглядностью, потому что все элементы здесь – артефакты. Расширение .class сообщает, что некоторые из артефактов представляют откомпилированные файлы классов Java, но понять, что META_INF – каталог, сложно. Это указывает на необходимость обо-

[1] Имеется в виду исполняемый файл jar (в Windows это jar.exe), поставляемый вместе с JDK. – *Примеч. науч. ред.*

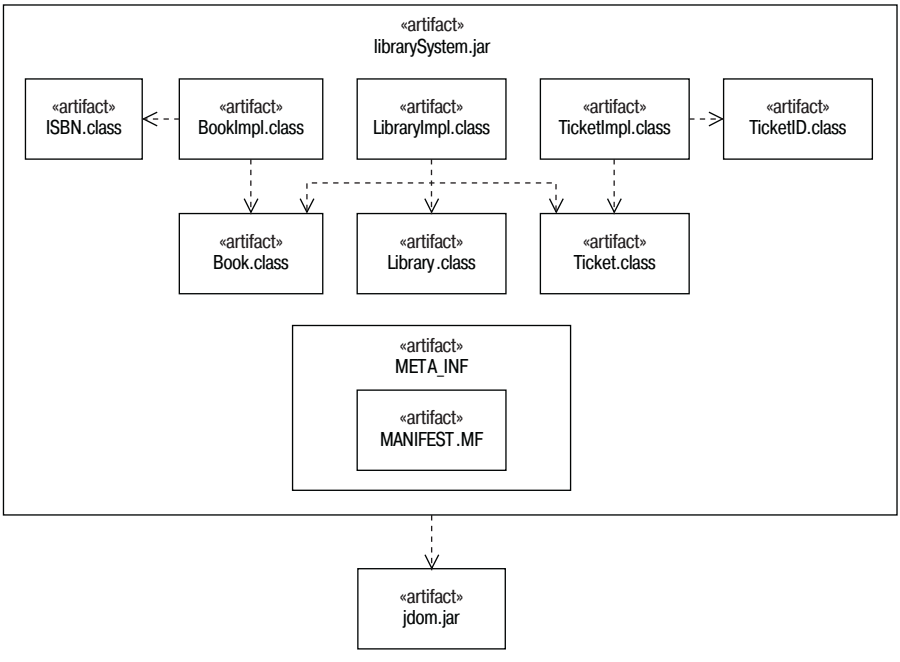


Рис. 24.7. *JAR-файл содержит Java-файлы классов для каждого класса и интерфейса системы*

значать артефакты стереотипами, чтобы было четко видно, что каждый из них представляет.

UML предлагает небольшое число стандартных стереотипов артефактов, представляющих разные типы файлов. Они перечислены в табл. 24.1.

Таблица 24.1

Стереотип артефакта	Семантика
«file»	Физический файл.
«deployment spec»	Описание деталей развертывания (например, web.xml в J2EE).
«document»	Универсальный файл, содержащий некоторую информацию.
«executable»	Исполняемый программный файл.
«library»	Статическая или динамическая библиотека, такая как динамически подключаемая библиотека (DLL) или файл Java-архива (JAR).
«script»	Сценарий, который может быть выполнен интерпретатором.
«source»	Исходный файл, который может быть скомпилирован в исполняемый файл.

Можно ожидать, что со временем будут разработаны различные UML-профили для конкретных программных платформ, таких как J2EE (Java 2 Platform, Enterprise Edition) и Microsoft .NET. Это обеспечит более богатый набор стереотипов артефактов (и других элементов). Спецификация UML 2.0 [UML2S] предоставляет примеры профилей для J2EE и EJB, Microsoft COM, Microsoft .NET и CORBA (Common Object Request Broker Architecture – общая архитектура посредника запросов к объектам).

В табл. 24.2 приведен пример профиля Java. Этого профиля недостаточно для моделирования приложений Java, поскольку в нем не хватает стереотипов для файлов классов и каталогов. Мы расширяем этот профиль двумя новыми стереотипами, представленными в табл. 24.3.

На рис. 24.8 расширенный Java-профиль из спецификации UML применен к нашей модели. Как видите, с использованием наглядных стереотипов диаграмма стала намного более выразительной.

Таблица 24.2

Стереотип	Применяется к	Семантика
«EJBEntityBean»	компоненту	Компонент-сущность EJB.
«EJBSessionBean»	компоненту	Сеансовый компонент EJB.
«EJBMessageDrivenBean»	компоненту	Управляемый сообщениями компонент EJB.
«EJBHome»	интерфейсу	«Домашний» интерфейс EJB.
«EJBRemote»	интерфейсу	Удаленный интерфейс EJB.
«EJBCreate»	операции	Операция создания EJB.
«EJBBusiness»	операции	Операция, поддерживающая бизнес-логику удаленного интерфейса EJB.
«EJBSecurityRoleRef»	ассоциации	Ассоциация между EJB и поставщиком, предоставляющим ссылку на роль в системе безопасности.
«EJBRoleName»	актеру	Имя роли в системе безопасности.
«EJBRoleNameRef»	актеру	Ссылка на роль в системе безопасности.
«JavaSourceFile»	артефакту	Исходный файл Java.
«JAR»	артефакту	Файл Java-архива.
«EJBQL»	выражению	Выражение на языке запросов EJB.

Таблица 24.3

Стереотип	Применяется к	Семантика
«JavaClassFile»	артефакту	Файл класса Java (откомпилированный исходный файл Java).
«directory»	артефакту	Каталог.

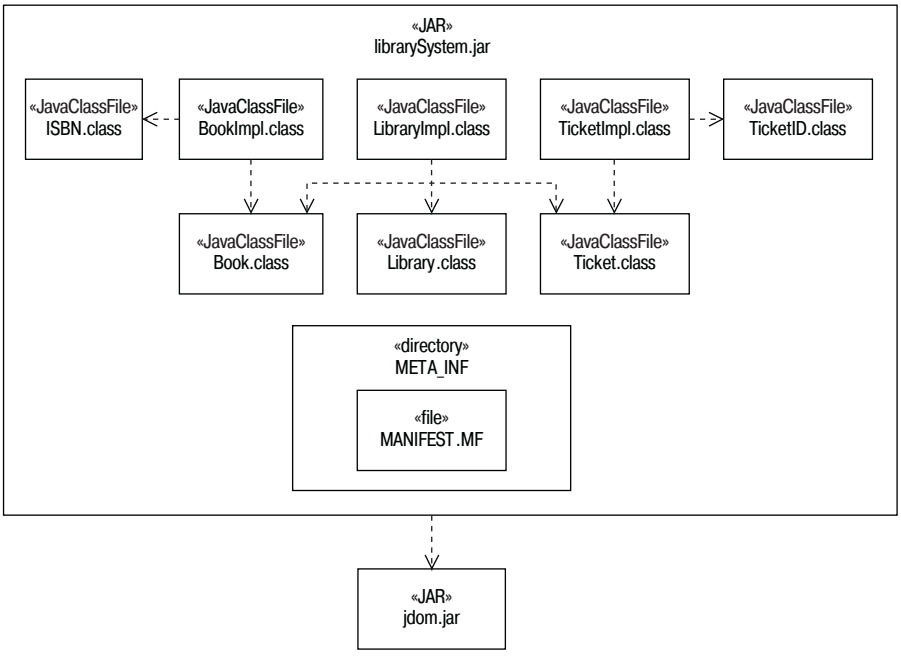


***Рис. 24.8.** К модели на рис. 24.7 применен расширенный Java-профиль из спецификации UML*

24.6. Развертывание

Простая экземплярная форма диаграммы развертывания представлена на рис. 24.9.

Данный пример взят из обучающего курса Java (*www.java.sun.com*). Это приложение преобразователя валют. На рис. 24.9 показан файл архива корпоративных приложений (Enterprise application Archive, EAR) ConverterApp.ear, развернутый в среде выполнения J2EE Server на узле server (сервер) типа WindowsPC. J2EE Server – это сервер приложений корпорации Sun, поставляемый как часть J2EE. EAR-файлы – особый тип JAR-файлов, в которых содержатся приложения J2EE. Развернутое серверное приложение используется клиентским приложением ConverterClient.jar, которое исполняется на узле client (клиент) типа WindowsPC.

Спецификацию развертывания можно прикрепить к развернутому артефакту, как показано на рисунке. В ней содержится основная информация о развертывании. В данном случае определены три вещи:

- EnterpriseBeanClass (класс корпоративного компонента) – класс, содержащий логику компонента;
- EnterpriseBeanName – имя корпоративного компонента, которое может применяться клиентом для организации доступа к компоненту;

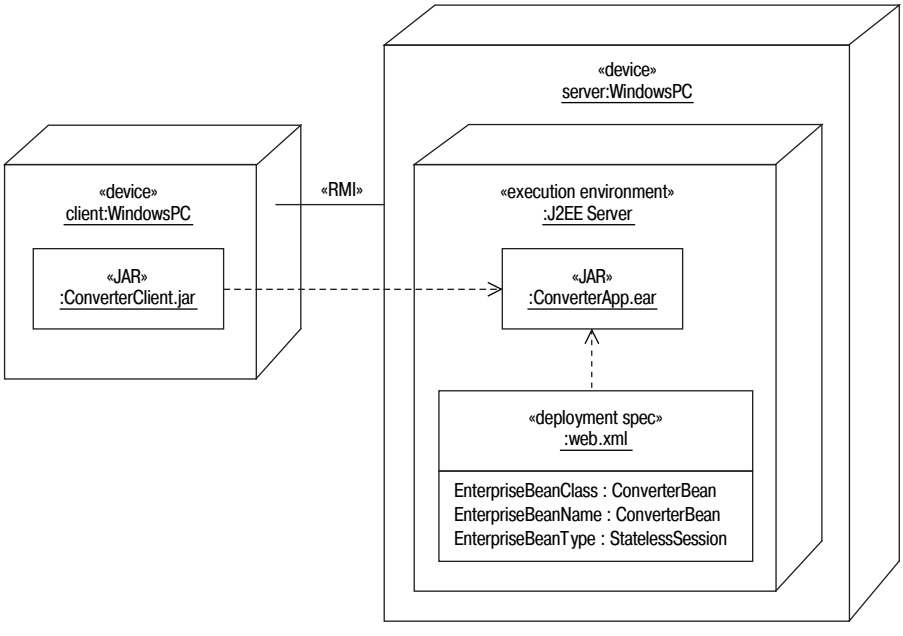


Рис. 24.9. *Экземплярная форма диаграммы развертывания для приложения преобразователя валют*

- EnterpriseBeanType (тип корпоративного компонента) – тип компонента. В данном случае это не имеющий состояния сеансовый компонент – компонент, используемый для простых транзакций; он не имеет состояний и не является сохраняемым.

Развертывание EJB – намного более сложный процесс, чем мог бы предложить этот простой пример, и некоторые из его деталей даже зависят от среды выполнения. Однако цель моделирования в развертывании – отражение основных моментов развертывания, поэтому представленного дескриптора может быть вполне достаточно.

24.7. Что мы узнали

Диаграммы развертывания позволяют моделировать распределение программной системы на физическом оборудовании. Мы узнали следующее:

- Деятельность UP Реализация архитектуры состоит в определении важных с архитектурной точки зрения компонентов и проецировании их на физическое оборудование.
- Диаграмма развертывания проецирует программную архитектуру на аппаратную архитектуру.

- В процессе проектирования можно создать «первое приближение» диаграммы развертывания, определяя узлы или экземпляры узлов и отношения. В процессе реализации это приближение дополняется компонентами или экземплярами компонентов.
- Дескрипторная форма диаграммы развертывания может использоваться для моделирования того, какие типы оборудования, программного обеспечения и соединений будут присутствовать в окончательно развернутой системе.
  - Описывает полный набор возможных сценариев развертывания.
  - Показывает:
    - узлы – на каких типах оборудования выполняется система;
    - отношения – типы соединений между узлами;
    - компоненты – типы компонентов, развернутых на конкретных узлах.
- Экземплярная форма диаграммы развертывания представляет конкретное развертывание системы на определенных, идентифицируемых частях оборудования.
  - Описывает один определенный вариант развертывания системы, возможно, на конкретном пользовательском сайте.
  - Показывает:
    - экземпляры узлов – конкретные части оборудования;
    - экземпляры отношений – конкретные отношения между экземплярами узлов;
    - экземпляры компонентов – конкретные идентифицируемые части программного обеспечения, развернутые на экземпляре узла, например конкретная копия Microsoft Office с уникальным серийным номером.
- Узел – представляет тип вычислительного ресурса.
  - «device» – тип физического устройства, такой как ПК или сервер Fire корпорации Sun.
  - «execution environment» – тип среды выполнения программного обеспечения, например веб-сервер Apache.
- Экземпляр узла – представляет конкретный вычислительный ресурс.
- Артефакт – представляет описание реальной сущности, такой как конкретный исполняемый файл.
  - Артефакты могут представлять один или более компонентов.
- Экземпляр артефакта – представляет определенный экземпляр конкретного артефакта, например определенную копию конкретного исполняемого файла, установленную на конкретном компьютере.

VI

Дополнительные материалы

25

Введение в OCL

25.1. План главы

Сначала мы собирались дать *очень* краткое введение в OCL, в основном охватывающее информацию, необходимую для сертификации по UML. Однако чем больше мы знакомились с существующей литературой по OCL, тем более ощущалась необходимость в простом, но исчерпывающем описании языка, ориентированном именно на ОО аналитика/проектировщика. Именно такое описание мы постарались представить в этой главе. Оно основывается на спецификации «Unified Modeling Language: OCL, version 2.0» [OCL1], но пока книга готовилась к изданию, в спецификацию могли быть внесены некоторые незначительные изменения.

Как ОО аналитик или проектировщик вы, вероятно, не очень хорошо знакомы с OCL. Надеемся, что прочитав главу, вы получите достаточно полное представление об OCL и сможете оценить те возможности, которые он предоставляет для подробного UML-моделирования.

Глава содержит довольно много материала, поэтому в ее план включены только основные темы (рис. 25.1).

25.2. Что такое объектный язык ограничений (OCL)?

> OCL может определять запросы, ограничения и операции запросов.

OCL – язык, позволяющий вводить в UML-модель дополнительную информацию. Это стандартное расширение UML, предоставляющее следующие возможности:

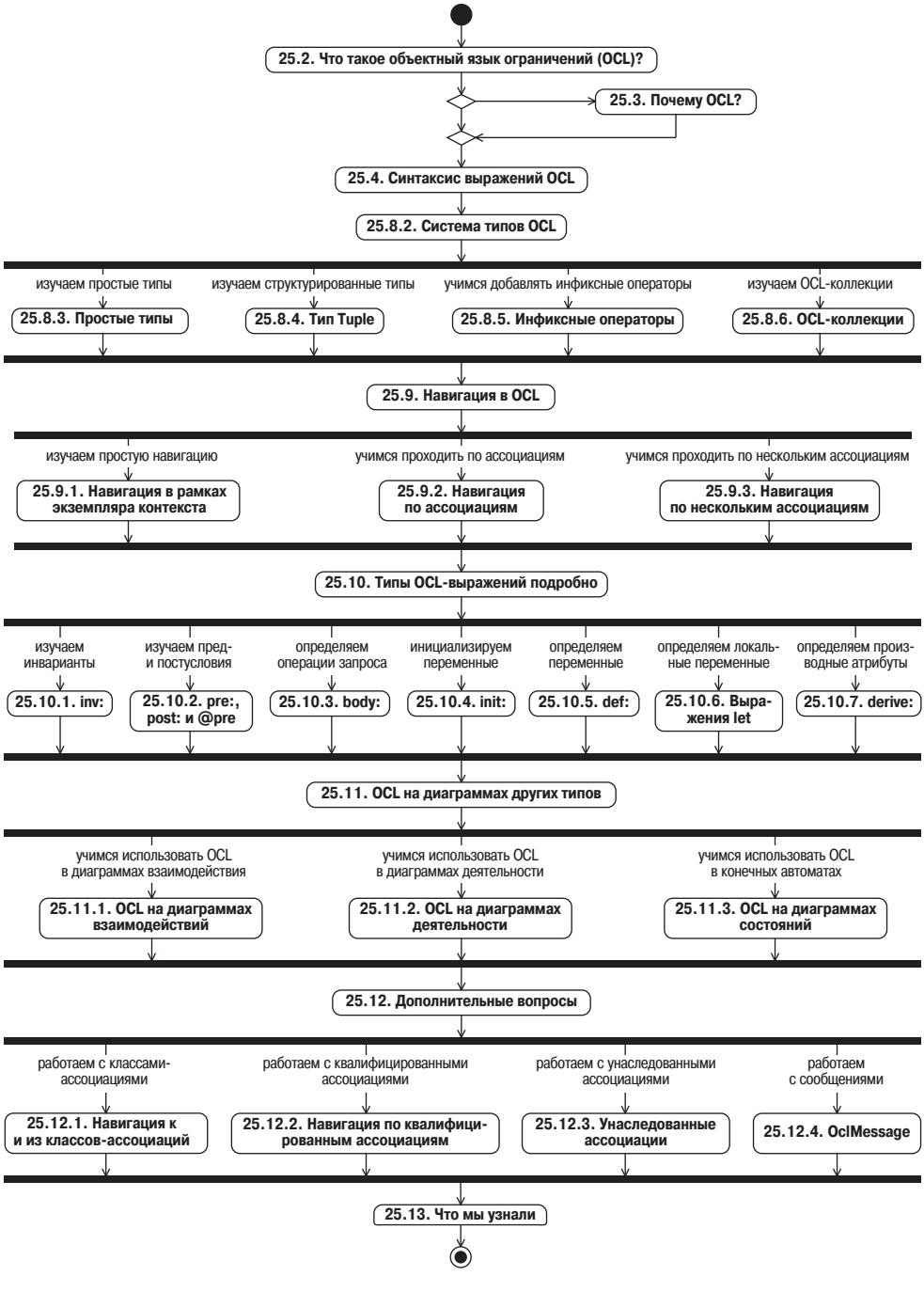


Рис. 25.1. *План главы*

- писать запросы для организации доступа к элементам модели и их значениям; это язык запросов, немного похожий на SQL;
- накладывать ограничения на элементы модели – можно определять бизнес-правила как ограничения, накладываемые на элементы модели;
- определять операции запросов.

> OCL не язык действий для UML, потому что выражения OCL не оказывают побочных эффектов.

Очень важно понять, что OCL не является языком действий для UML. С его помощью невозможно определить поведение, потому что OCL-выражения *не имеют побочных эффектов*. Таким образом:

- OCL не может менять значение элемента модели – можно только запрашивать значения и накладывать условия на значения;
- OCL не может определять операции, за исключением операций запросов;
- OCL может выполнять только операции запроса, не изменяющие значений;
- OCL не может применяться для динамического определения бизнес-правил во время выполнения – с его помощью можно описывать бизнес-правила только во время моделирования.

Выражения OCL могут храниться в файлах, ассоциированных с UML-моделью. Как это делается, зависит от конкретного используемого средства моделирования. Однако спецификация OCL определяет основанный на XML формат обмена, поэтому OCL-выражения можно использовать с разными инструментальными средствами.

OCL-выражения также могут быть прикреплены непосредственно к элементам UML-модели в виде примечаний. Преимущество такого подхода состоит в визуализации OCL-выражений в модели. Но есть и недостаток – возможность загромождения модели, если выражений очень много.

25.3. Почему OCL?

Есть несколько причин, по которым OCL может оказаться полезным.

- OCL позволяет средствам моделирования при наличии соответствующей поддержки анализировать UML-модели – например, сюда может входить проверка согласованности.
- OCL позволяет средствам моделирования, имеющим поддержку OCL, генерировать код на основании OCL-выражений – например, с помощью инструментального средства можно было бы генерировать код реализации OCL-ограничений, таких как предусловия и постусловия операции.

- OCL обеспечивает возможность создавать более точные модели – это уменьшает вероятность неверного толкования модели.
- OCL является частью сертификационного экзамена OCUP (OMG Certified UML Professional – сертифицированный OMG специалист UML) – в тест включено очень мало вопросов по OCL, но они там есть.

Есть несколько причин, по которым OCL может показаться *бесполезным*.

- OCL довольно сложен для восприятия – он имеет нестандартный синтаксис и множество необычных «сокращенных» форм.
- В настоящее время с OCL знакомы очень немногие разработчики моделей и еще меньшее число программистов – таким образом, может оказаться, что OCL-выражения не «найдут своего зрителя».
- Такой уровень точности, который предлагает OCL, может оказаться ненужным – например, если создается неформальная UML-модель, которая будет передана для уточнения программистам, OCL может быть излишним.

Мы считаем, что OCL – это просто еще один инструмент в инструментарии средств моделирования, который может использоваться при создании строгих UML-моделей. Его полезно знать, чтобы уметь применять там, где он действительно нужен.

25.4. Синтаксис выражений OCL

OCL – небольшой язык, но он обладает на удивление сложным синтаксисом, который до сих пор развивается. В частности, он имеет синтаксические исключения и сокращения, которые иногда могут сбить с толку несведущего пользователя. Синтаксис имеет много общего со стилем C/C++/Java с некоторыми элементами Smalltalk.

Семантика языка OCL (формально определенная) не зависит от какого-либо конкретного синтаксиса. Со временем это может привести к возникновению альтернативных синтаксисов OCL. Уже существует SQL-подобный синтаксис OCL для бизнес-моделирования, описание которого можно найти в книге [Warmer 1].

В этой главе мы потратили много сил, пытаясь выделить наиболее непонятные аспекты синтаксиса (и семантики) OCL и сделать их максимально прозрачными!

В отличие от большинства основных языков программирования, OCL является декларативным языком. Это означает, что описывается результат, который необходимо получить, а не способ достижения этого результата. Такие языки программирования, как Java, C#, C++ и большинство других, являются процедурными – в них шаг за шагом описывается, как получается желаемый результат.

В традиционных языках программирования создаются программы, исполняющиеся для предоставления пользователю некоторого результата.

> OCL-выражения прикрепляются к элементам UML-модели.

В OCL создаются выражения, которые прикрепляются к элементам UML-модели и определяют или ограничивают ее некоторым образом. Это фундаментальный момент и, вероятно, самый большой камень преткновения, с которым сталкиваются разработчики моделей и программисты при первой встрече с OCL. OCL *не* язык программирования, это язык ограничений. Самое главное, что необходимо помнить об OCL, – здесь определяются запросы и условия, а *не* поведение.

Общая форма OCL-выражения представлена на рис. 25.2.

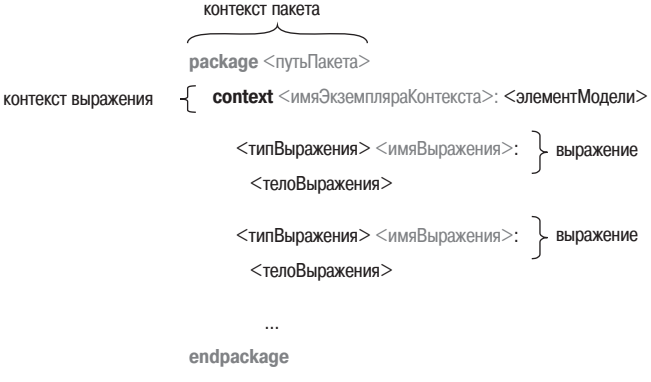


***Рис. 25.2.** Общая форма OCL-выражения*

На рисунке **полужирным шрифтом** выделено ключевое слово OCL, а серым – необязательные элементы. Угловые скобки (**<…>**) указывают места, которые должны быть заменены соответствующими значениями.

> У каждого OCL-выражения есть значение.

OCL – типизированный язык. Каждое OCL-выражение приводится к объекту некоторого типа. Как видно из рис. 25.2, OCL-выражения можно разбить три части:

- контекст пакета (необязательный);
- контекст выражения (обязательный);
- одно или более выражений.

В следующих разделах мы подробно рассмотрим каждую из этих частей. Для этого воспользуемся моделью представленной на рис. 25.3. Она предоставляет контекст для наших примеров OCL-выражений.

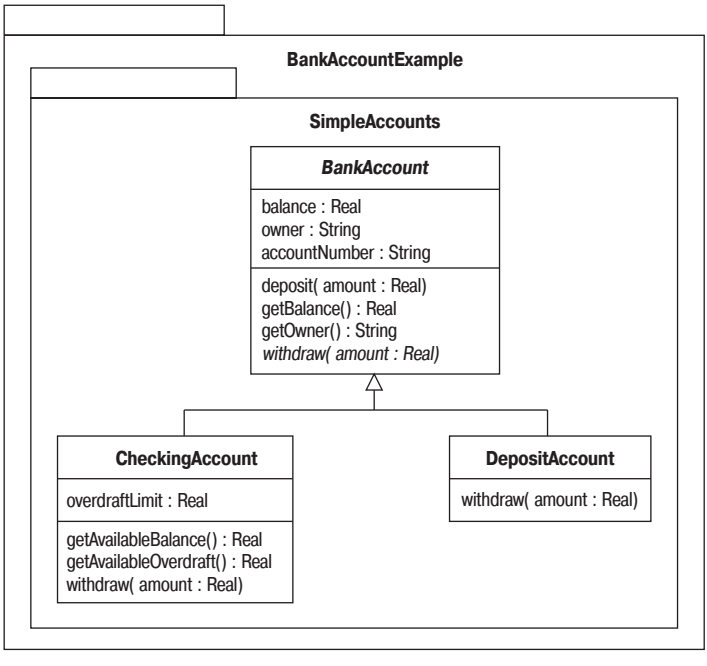


Рис. 25.3. *Модель с OCL-выражениями*

25.5. Контекст пакета и составные имена

> Контекст пакета определяет пространство имен OCL-выражения.

Необязательный контекст пакета позволяет задавать пакет, определяющий пространство имен OCL-выражения. Контекст пакета подчиняется следующим правилам:

- Если контекст пакета *не* задан, пространством имен выражения по умолчанию становится вся модель.
- Если OCL-выражение прикрепляется непосредственно к элементу модели, по умолчанию пространством имен этого выражения становится пакет, которому принадлежит этот элемент.

Например, на рис. 25.3 контекст пакета можно было бы определить как:

```
package BankAccountExample::SimpleAccounts
...
endpackage
```

Если все элементы UML-модели имеют уникальные имена, нет необходимости в использовании контекста пакета, поскольку на каждый эле-

мент можно однозначно сослаться по имени. Однако если в *разных* пакетах есть элементы с *одинаковыми* именами, можно:

- для каждого OCL-выражения, ссылающегося на любой из элементов, определить контекст пакета ИЛИ (OR)
- ссылаться на элементы с помощью полных составных имен, например:

```
BankAccountExample::SimpleAccounts::BankAccount.
```

OCL-синтаксис имени пути:

```
Пакет1::Пакет2:: … ::ПакетN::ИмяЭлемента
```

25.6. Контекст выражения

> Контекст выражения обозначает элемент UML-модели, к которому прикреплено OCL-выражение.

Контекст выражения обозначает элемент UML-модели, к которому прикреплено OCL-выражение.

Например, если требуется прикрепить OCL-выражение к классу CheckingAccount на рис. 25.3, контекст выражения можно было бы определить следующим образом:

```
package BankAccountExample::SimpleAccounts
 context account:CheckingAccount
…
endpackage
```

> OCL-выражения должны записываться в рамках экземпляра контекста.

Контекст выражения определяет *экземпляр контекста* (*contextual instance*), имеющий необязательное имя (account) и обязательный тип (CheckingAccount).

Экземпляр контекста необходимо рассматривать как образец экземпляра класса, который может использоваться в OCL-выражениях.

Если экземпляру контекста присвоено имя, его можно использовать в теле выражения для ссылки на экземпляр контекста. Если имя экземпляра контекста *не* задано, сослаться на него можно с помощью ключевого слова OCL self. Обычно мы используем это ключевое слово.

В приведенном выше выражении экземпляр контекста – это экземпляр класса CheckingAccount. Ссылаться на него можно как по имени (account), так и с помощью ключевого слова self.

Тип экземпляра контекста зависит от контекста выражения.

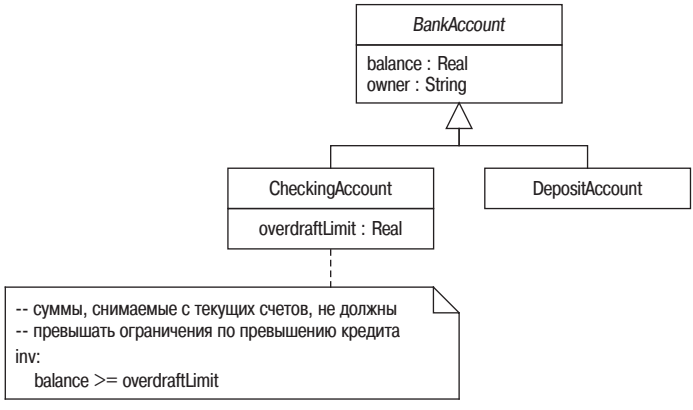


***Рис. 25.4.** OCL-выражение прикреплено к элементу модели как примечание*

- Если контекст выражения – классификатор, экземпляр контекста всегда является экземпляром этого классификатора.
- Если контекст выражения – операция или атрибут, экземпляр контекста обычно является экземпляром классификатора, которому *принадлежат* эти операция или атрибут.

Когда OCL-выражение прикрепляется к элементу модели как примечание (рис. 25.4), контекст выражения определяется точкой присоединения примечания, поэтому нет необходимости задавать его явно.

25.7. Типы OCL-выражений

> Существует две категории OCL-выражений – ограничивающие и определяющие.

Существует восемь разных типов OCL-выражений, все они представлены в табл. 25.1. Можно заметить, что эти выражения подразделяются на две категории: те, которые задают ограничения (inv:, pre: и post:), и те, которые определяют атрибуты, тела операций и локальные переменные (init:, body:, def:, let и derive:).

Таблица 25.1

Тип выражения	Синтаксис	Применяется к	Экземпляр контекста	Семантика	Раздел
Операции, которые ограничивают					
Инвариант	inv:	Классификатор	Экземпляр классификатора	Инвариант должен быть истинным для всех экземпляров классификатора.	25.10.1

Тип выражения	Синтаксис	Применяется к	Экземпляр контекста	Семантика	Раздел
Предусловие	pre:	Операция Элемент поведения	Экземпляр классификатора, которому принадлежит операция	Предусловие должно быть истинным, чтобы операция могла выполняться.	25.10.2
Постусловие	post:	Операция Элемент поведения	Экземпляр классификатора, которому принадлежит операция	Постусловие должно быть истинным после выполнения операции. Ключевое слово result ссылается на результат операции.	25.10.2
Операции, которые определяют					
Тело операции запроса	body:	Операция запроса	Экземпляр классификатора, которому принадлежит операция	Определяет тело операции запроса.	25.10.3
Начальное значение	init:	Атрибут Конец ассоциации	Атрибут Конец ассоциации	Определяет начальное значение атрибута или конец ассоциации.	25.10.4
Определение	def:	Классификатор	Экземпляр классификатора, которому принадлежит операция	Добавляет переменные или вспомогательные операции в контекстный классификатор. Используется в OCL-выражениях контекстного классификатора.	25.10.5
Присваивание	let	OCL-выражение	Экземпляр контекста OCL-выражения	Добавляет локальные переменные в OCL-выражения.	25.10.6
Производное значение	derive:	Атрибут Конец ассоциации	Атрибут Конец ассоциации	Определяет правило вывода для производных атрибутов или конца ассоциации.	25.10.7

Ограничивающим операциям (inv:, pre: и post:) можно присвоить имя выражения (expressionName). Это позволяет ссылаться на них по имени, например, чтобы связать их с описаниями прецедентов или другими документами требований. Хорошим OCL-стилем считается:

- всегда давать имя ограничениям (даже несмотря на то, что имя является необязательным);

- выбирать описательные имена, отражающие семантику ограничения;
- гарантировать уникальность имен ограничений в рамках модели;
- имена ограничений записывать в стиле lowerCamelCase.

Определяющим операциям (init:, body:, def:, let, derive:) имя выражения присвоить *нельзя*.

Семантика различных типов OCL-выражений подробно рассматривается в разделе 25.10 сразу после обсуждения тела и синтаксиса OCL-выражений.

25.8. Тело выражения

Тело выражения содержит суть OCL-выражения. Простой пример представлен на рис. 25.4.

В следующих разделах мы представляем синтаксис OCL, чтобы дать вам возможность самим научиться создавать тело OCL-выражений.

25.8.1. Комментарии, ключевые слова и правила старшинства операций

Процессоры OCL игнорируют комментарии. Комментарии необходимо активно использовать, чтобы сделать OCL-выражения более понятными.

> Комментарии необходимо активно использовать, чтобы сделать OCL-выражения более понятными.

Вы получите хороший комментарий OCL-выражения, если просто перепишете это выражение на английском (или немецком, или любом родном вам языке). В этой главе будут приведены примеры того, как это делается.

В OCL используется два стиля комментирования:

```
-- Это однострочный комментарий. Игнорируется вся строка после знака --.

/* Это многострочный комментарий.
Игнорируется все, что заключено в символы комментария. */
```

Мы предпочитаем использовать однострочные комментарии.

В OCL есть очень небольшой набор ключевых слов, которые *не могут* использоваться как имена в OCL-выражениях:

```
and, attr, context, def, else, endif, endpackage, if, implies, in, inv, let, not, oper, or, package,
post, pre, then, xor, body, init, derive.
```

Все они рассматриваются в этой главе.

Операции OCL подчиняются правилам старшинства, как показано на рис. 25.5.

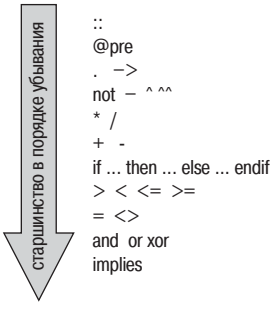


Рис. 25.5. *Приоритет операций*

В любом OCL-выражении более высокие по старшинству операции выполняются первыми. Таким образом, например, выражение

```
1 + 2 * 3
```

в результате дает 7, потому что операция умножения * имеет более высокий приоритет, чем операция сложения +.

Приоритет операций можно изменить с помощью скобок, тогда

```
(1 + 2) * 3
```

в результате дает 9.

В *любом* языке программирования всегда лучше использовать скобки, а не полагаться на правила старшинства операций!

25.8.2. Система типов OCL

OCL – строго типизированный язык. Для написания OCL-выражений необходимо понимать систему типов OCL.

> Простые типы OCL – это Boolean, Integer, String и Real.

Все языки программирования имеют набор простых типов, и OCL не исключение.

Простые типы – это Boolean, Integer, Real и String. Они обсуждаются в разделе 25.8.3. В OCL также есть структурированный тип, Tuple (кортеж), который рассматривается в разделе 25.8.4.

Кроме простых типов и Tuple OCL имеет ряд встроенных типов, которые перечислены ниже.

- OclAny – супертип всех типов OCL и ассоциированной UML-модели;
- OclType – подкласс OclAny – перечисление всех типов ассоциированной UML-модели;
- OclState – подкласс OclAny – перечисление всех состояний ассоциированной UML-модели;

- OclVoid – тип «null» в OCL – имеет единственный экземпляр под названием OclUndefined;
- OclMessage – представляет сообщение (раздел 25.12.4).

> Все классификаторы UML-модели доступны OCL-выражениям.

Необычный, но ключевой аспект системы типов OCL состоит в том, что *все* классификаторы ассоциированной UML-модели становятся типами в OCL. Это означает, что OCL-выражения могут *напрямую* ссылаться на классификаторы ассоциированной модели. Это то, что позволяет OCL выполнять роль языка ограничений.

> **OclAny** – супертип всех типов OCL.

В OCL *все* типы – это подтипы OclAny. Простые типы являются прямыми подтипами OclAny, тогда как типы UML-модели являются подклассами OCLType, который, в свою очередь, является подклассом OclAny. Каждый тип наследует небольшой набор полезных операций (табл. 25.2).

Таблица 25.2

Операция OclAny	Семантика
Операции сравнения	
a = b	Возвращает true, если a – тот же объект, что и b, в противном случае возвращает false.
a <> b	Возвращает true, если a *не* тот же объект, что и b, в противном случае возвращает false.
a.oclIsTypeOf(b : OclType) : Boolean	Возвращает true, если a того же типа, что и b, в противном случае возвращает false.
a.oclIsKindOf(b : OclType) : Boolean	Возвращает true, если a того же типа, что и b, или a – подтип b.
a.oclInState(b : OclState) : Boolean	Возвращает true, если a находится в таком же состоянии, что и b, в противном случае возвращает false.
a.oclIsUndefined() : Boolean	Возвращает true, если a = OclUndefined.
Операции запроса	
A::allInstances( ) : Set(A)	Это операция уровня класса, возвращающая Set всех экземпляров типа A.
a.oclIsNew( ) : Boolean	Возвращает true, если a был создан в ходе выполнения операции. Может использоваться только в постусловиях операции.

Операция OclAny	Семантика
Операции преобразования	
a.oclAsType(SubType) : SubType	Возвращает результат приведения a к типу SubType. Это операция приведения, и объект a может быть приведен только к одному из его подтипов или супертипов. Приведение к супертипу обеспечивает доступ к переопределенным возможностям супертипа.

Вероятно, самая необычная операция OclAny – allInstances() (все экземпляры). Это операция уровня класса (применяется непосредственно к классу, а не к конкретному экземпляру), она возвращает Set всех экземпляров данного класса, существующих на момент вызова этой операции. Ни в одном из широко используемых языков программирования нет такой встроенной возможности, поэтому спецификация OCL определяет allInstances() как необязательную для реализации операцию в инструментальных средствах, работающих с OCL. Это означает, что ваш инструмент OCL, возможно, не сможет интерпретировать выражения, использующие allInstances().

В начале может показаться странным, что все типы, тщательно определенные в UML-модели, при использовании в OCL-выражениях автоматически получают новый супертип, OclType. Однако OCL был вынужден сделать это, чтобы обеспечить общий объектный протокол (определенный OclAny), который он может использовать для работы с типами.

25.8.3. Простые типы

Простые типы OCL – это Boolean, Integer, Real и String. Их семантика во многом аналогична семантике этих типов в других языках программирования (табл. 25.3).

Таблица 25.3

Базовый тип OCL	Семантика
Boolean	Может принимать значения true или false
Integer	Целое число
Real	Число с плавающей точкой
String	Последовательность символов Строковые литералы заключаются в одинарные кавычки, например 'Jim'

Поскольку OCL является языком моделирования, а не языком программирования, его спецификация *не налагает ограничений* на длину строк (String), размер целых (Integer), размер и точность действительных чисел (Real).

25.8.3.1. Тип Boolean

Тип Boolean принимает два значения: true и false. В нем имеется набор операций, возвращающих значения типа Boolean. Двоичные операции приведены в табл. 25.4. В данной таблице истинности представлены результаты логических операций над значениями a и b.

Таблица 25.4

a	b	a = b	a<>b	a.and(b)	a.or(b)	a.xor(b)	a.implies(b)
true	true	true	false	true	true	false	true
true	false	false	true	false	true	true	false
false	true	false	true	false	true	true	true
false	false	true	false	false	false	false	true

Все эти операции должны быть знакомы вам из других языков программирования, кроме операции implies (импликация). Она пришла из формальной логики и состоит из предпосылки, a, и следствия, b. Результат операции принимает значение true, если предпосылка и следствие имеют одно и то же значение или если предпосылка принимает значение false, а следствие – true. Операция принимает значение false, если предпосылка принимает значение true, а следствие – false.

Существует также унарный оператор not, представленный в табл. 25.5.

Таблица 25.5

a	not a
true	false
false	true

Логические выражения часто используются в выражениях if...then... else в соответствии со следующим синтаксисом:

```
if <логическоеВыражение> then
  <oclВыражение1>
  else
  <oclВыражение2>
endif
```

25.8.3.2. Типы Integer и Real

Тип Integer представляет целое число, Real – число с плавающей точкой. Длина целых (Integer) и длина и точность действительных чисел (Real) не ограничены. Integer и Real имеют обычный набор инфиксных арифметических операций со стандартной семантикой:

```
=, <>, <, >, <=, >=, +, -, *, /
```

Также эти типы имеют операции, описанные в табл. 25.6.

Таблица 25.6

Синтаксис	Семантика	Применяется к
a.mod(b)	Возвращает остаток от деления a на b например a = 3, b = 2, a.mod(b) возвращает 1	Integer
a.div(b)	Возвращает лишь целую часть от деления a на b например a = 8, b = 3, a.div(b) возвращает 2	Integer
a.abs()	Возвращает положительное a например a = (-3), a.abs() возвращает 3	Integer и Real
a.max(b)	Возвращает большее из чисел a и b например a = 2, b = 3, a.max(b) возвращает b	Integer и Real
a.min(b)	Возвращает меньшее из чисел a и b например a = 2, b = 3, a.min(b) возвращает a	Integer и Real
a.round()	Возвращает Integer, ближайшее к a Если два целых одинаково близки, возвращается большее из них например a = 2.5, a.round() возвращает 3, а не 2 a = (-2.5), a.round() возвращает -2, а не -3	Real
a.floor()	Возвращает ближайшее Integer, которое меньше или равно a например a = 2.5, a.floor() возвращает 2 a = (-2.5), a.floor() возвращает -3	Real

25.8.3.3. Тип String

Строковые операции OCL (табл. 25.7) опять-таки довольно стандартны, аналогичный набор можно найти практически в любом языке программирования.

Таблица 25.7

Синтаксис	Семантика
s1 = s2	Возвращает true, если последовательность символов s1 соответствует последовательности символов s2, в противном случае возвращает false
s1 <> s2	Возвращает true, если последовательность символов s1 не соответствует последовательности символов s2, в противном случае возвращает false
s1.concat(s2)	Возвращает новую String, являющуюся объединением s1 и s2 например 'Jim'.concat(' Arlow') возвращает 'Jim Arlow'
s1.size()	Возвращает число символов (Integer) в s1 например 'Jim'.size() возвращает 3
s1.toLower()	Возвращает новую строку символов (String), записанных в нижнем регистре например 'Jim'.toLower() возвращает 'jim'

Таблица 25.7 (продолжение)

Синтаксис	Семантика
s1.toUpper()	Возвращает строку символов (String), записанных в верхнем регистре например 'Jim'.toUpper() возвращает 'JIM'
s1.toInteger()	Преобразует s1 в значение типа Integer например '2'.toInteger() возвращает 2
s1.toReal()	Преобразует s1 в значение типа Real например '2.5'.toReal() возвращает 2.5
s1.substring(start, end)	Возвращает новую String, являющуюся подстрокой s1, начинающуюся от символа, находящегося в позиции start и заканчивающуюся символом в позиции end Примечания: * start и end должны быть типа Integer * Первый символ в s1 имеет индекс 1 * Последний символ в s1 имеет индекс s1.size() например 'Jim Arlow'.substring(5, 9) возвращает 'Arlow'

Строки в OCL неизменны.

Строки в OCL неизменны (immutable). Это значит, что, будучи инициализированными, они не могут быть изменены. Такие операции, как s1.concat(s2), всегда возвращают новую строку String.

25.8.4. Тип Tuple

Объекты типа Tuple – это структурированные объекты, имеющие одну или более именованных частей.

Объекты типа Tuple (кортеж) – это структурированные объекты, имеющие одну или более именованных частей. Tuple необходимы, потому что некоторые операции OCL возвращают несколько объектов. Tuple имеют следующий синтаксис:

```
Tuple { имяЧасти1:типЧасти1 = значение1, имяЧасти2:типЧасти2 = значение2, ... }
```

Имя и значение каждой части обязательны, тип – необязателен. Порядок расположения частей не определен.

Вот пример Tuple, представляющего информацию об этой книге:

```
Tuple { title:String = 'UML 2 and the Unified Process', publisher:String = 'Addison Wesley' }
```

Части Tuple могут быть инициализированы любым допустимым OCL-выражением. В приведенном выше примере мы использовали строковые литералы.

Доступ к частям Tuple осуществляется с помощью оператора «точка». Например, следующее выражение возвращает значение 'Addison Wesley':

```
Tuple { title:String = 'UML 2 and the Unified Process', publisher:String =
'Addison Wesley' }.publisher
```

OCL – строго типизированный язык, поэтому каждый Tuple *должен* быть определенного типа. TupleType (тип кортежа) – это анонимный тип. Он не имеет имени и определяется *неявно* при создании Tuple. Однако тип TupleType можно задать *явно*. Например, TupleType для приведенного выше Tuple может быть записан в OCL так:

```
TupleType { title:String, publisher:String }
```

Обычно явное определение TupleType необходимо только в случае, если вы хотите создать коллекцию этого типа (раздел 25.8.6), например

```
Set( TupleType{ title:String, publisher:String} )  - - создает Set, который может хранить
                                                     - - объекты Tuple
```

25.8.5. Инфиксные операторы

Как вы увидите в нескольких последних разделах, операции, ассоциированные с простыми типами OCL, бывают двух видов. Есть синтаксис обычного вызова операции, например

```
a.toUpper()
```

и есть инфиксные операторы, в которых оператор располагается *между* операндами, например

```
a < b
```

Инфиксные операторы синтаксически более удобны. Вместо a.lessThan(b) записывается a < b. Такая форма более удобочитаема, особенно в сложных выражениях.

Инфиксные операторы также могут использоваться с типами из ассоциированной UML-модели *при условии* использования правильной сигнатуры операций. Приведенный на рис. 25.6 класс Money определяет некоторые логические и арифметические инфиксные операции.

Таким образом, если и a, и b имеют тип Money, можно использовать такие выражения

```
a < b
```

Однако обратите, пожалуйста, внимание, что спецификация OCL *запрещает* явные вызовы операций, такие как a.<(b), несмотря на то, что разумно было бы ожидать их наличия!

25.8.6. OCL-коллекции

OCL предоставляет довольно обширный набор типов коллекций, которые могут хранить другие объекты, включая и другие коллекции.

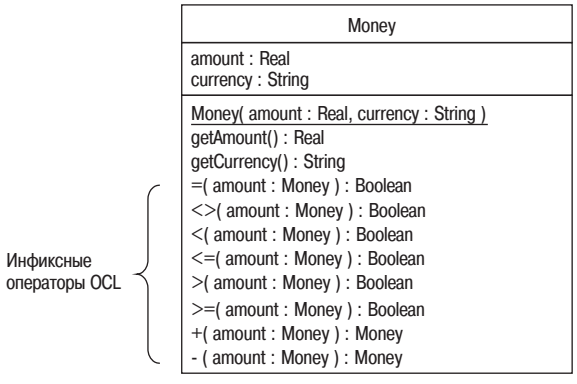


Рис. 25.6. *Инфиксные операторы OCL*

OCL-коллекции неизменны. Это означает, что операции над коллекциями *не* меняют их состояния. Например, при вызове операции для добавления или удаления элемента коллекции эта операция возвращает *новую* коллекцию, а исходная коллекция остается неизменной.

Мы уже говорили о типах коллекций OCL в разделе 18.10. Их семантика сведена в табл. 25.8. Обратите внимание на соответствие каждого из типов коллекций OCL паре свойств конца ассоциации. Применяемые по умолчанию свойства конца ассоциации – { unordered, unique }.

Таблица 25.8

Коллекция OCL	**Упорядоченность**	**Уникальность (дублирования не допускаются)**	**Свойства конца ассоциации**
Set	Нет	Да	{ unordered, unique } – применяются по умолчанию
OrderedSet	Да	Да	{ ordered, unique }
Bag	Нет	Нет	{ unordered, nonunique }
Sequence	Да	Нет	{ ordered, nonunique }

OCL-коллекции – это фактически шаблоны (раздел 17.7), для использования которых необходимо создать их экземпляр определенного типа. Например, OCL-выражение

```
Set( Customer )
```

создает экземпляр шаблона Set типа Customer. Тем самым определяется шаблон Set для хранения объектов типа Customer. Можно создавать экземпляры OCL-коллекций любого из доступных типов.

Коллекцию констант можно определить, просто перечислив в фигурных скобках ее элементы:

```
OrderedSet{ 'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday' }
```

При этом автоматически создается экземпляр коллекции, тип которого соответствует типу перечисленных элементов.

Последовательности (Sequence) целых (Integer) литералов имеют собственный специальный синтаксис, использующий *описание интервала* (*interval specification*):

```
<start>...<end>
```

Это означает: «все Integer между <start> и <end>», где <start> и <end> – OCL-выражения, результатами которых являются Integer. Например

```
Sequence{ 1 ... 7} эквивалентна Sequence{ 1, 2, 3, 4, 5, 6, 7 }
Sequence{ 2 ... ( 3 + 4 ) } эквивалентна Sequence{ 2, 3, 4, 5, 6, 7 }
```

Коллекции могут включать другие коллекции, например

```
OrderedSet{ OrderedSet{ 'Monday', 'Tuesday' }, OrderedSet{ 'Wednesday','Thursday', 'Friday' } }
```

25.8.6.1. Операции над коллекциями

> Любой единичный объект может быть интерпретирован как Set, содержащий только один элемент.

Коллекции обладают большим набором операций. Они *должны* инициироваться с помощью специального синтаксиса, в котором используется оператор «стрелка»:

```
коллекция->операцияКоллекции( параметры... )
```

Этот специальный синтаксис необходим, потому что OCL может интерпретировать любой единичный объект как Set, содержащий только один объект. Таким образом, если объект имеет, например, операцию count() и в Set есть операция с тем же именем count(), OCL нужно как-то различать эти две операции: принадлежащую объекту и принадлежащую коллекции. Это осуществляется путем использования оператора «точка» и вызова операций над коллекциями с помощью оператора «стрелка».

В следующих нескольких разделах рассматривается семантика операций над коллекциями. Чтобы было проще ссылаться на операции, мы разбили их на следующие категории:

- операции преобразования – преобразовывают один тип коллекции в другой (раздел 25.8.6.2);
- операции сравнения – сравнивают коллекции (раздел 25.8.6.3);
- операции запроса – получают информацию о коллекции (раздел 25.8.6.4);
- операции доступа – обеспечивают доступ к элементам коллекции (раздел 25.8.6.5);
- операции выбора – возвращают новую коллекцию, содержащую подмножество или надмножество коллекции (раздел 25.8.6.6).

Кроме того, OCL-коллекции обладают полным набором итерационных операций. Они являются довольно сложными и имеют необычный синтаксис, поэтому обсуждаются отдельно в разделе 25.8.7.

Мы также ввели несколько обозначений, чтобы упростить обсуждение коллекций и сделать его более компактным:

- X(T) – краткая запись, где X может быть Set, OrderedSet, Bag или Sequence;
- целевая коллекция – объект, операция которого вызывается.

При чтении следующих разделов необходимо помнить, что типы коллекций являются *шаблонными* типами. Это означает, что

Set(T) – это экземпляр Set типа T.

Таким образом, X(T) представляет экземпляры Set, OrderedSet, Bag или Sequence для элементов типа T.

25.8.6.2. Операции преобразования

Операции преобразования (табл. 25.9) преобразуют коллекцию одного типа в другой, возвращая новую коллекцию требуемого типа.

Например

```
Bag{ 'Homer', 'Meg' }->asOrderedSet()
```

возвращает новый OrderedSet, содержащий объекты 'Homer' и 'Meg' типа String.

Таблица 25.9

Операции преобразования **Операции над коллекциями**	**Семантика**
X(T)::asSet() : Set(T) X(T)::asOrderedSet() : OrderedSet(T) X(T)::asBag() : Bag(T) X(T)::asSequence() : Sequence(T)	Преобразует коллекцию одного типа в коллекцию другого типа. При преобразовании коллекции в тип Set дублирующиеся элементы отбрасываются. При преобразовании коллекции в OrderedSet или Sequence сохраняется исходный порядок (если таковой имеется), в противном случае устанавливается произвольный порядок.
X(T)::flatten() : X(T2)	В результате получается новая плоская коллекция с элементами типа T2. Например, если имеется: Set{ Sequence{ 'A', 'B' }, Sequence{ 'C', 'D' } } Set содержит экземпляры типа Sequence, которые содержат экземпляры типа String – результатом применения к Set оператора flatten является множество строк (Set содержащий элементы типа String).

Учитываются ограничения и исходной, и результирующей коллекции. В данном случае исходная коллекция является неупорядоченной. Но результирующая коллекция упорядочена, поскольку операция закрепляет этот произвольный порядок для результирующей коллекции.

25.8.6.3. Операции сравнения

Операции сравнения (табл. 25.10) сравнивают целевую коллекцию с коллекцией-параметром того же типа и возвращают результат типа Boolean. Эти операции учитывают ограничения упорядочения коллекций.

Таблица 25.10

Операции сравнения **Операция над коллекциями**	**Семантика**
X(T)::=(y : X(T)) : Boolean	Set и Bag возвращают true, если в y содержатся такие же элементы, как в целевой коллекции. OrderedSet и Sequence возвращают true, если в y содержатся такие же элементы и в том же порядке, как в целевой коллекции.
X(T)::<>(y : X(T)) : Boolean	Set и Bag возвращают true, если в y *не* содержатся такие же элементы, как в целевой коллекции. OrderedSet и Sequence возвращают true, если в y *не* содержатся такие же элементы, расположенные в том же порядке, как в целевой коллекции.

25.8.6.4. Операции запроса

Операции запроса (табл. 25.11) позволяют получить информацию о коллекции.

Таблица 25.11

Операции запроса **Операция над коллекциями**	**Семантика**
X(T)::size() : Integer	Возвращает количество элементов в целевой коллекции
X(T)::sum() : T	Возвращает сумму всех элементов целевой коллекции Тип T *должен* поддерживать оператор +
X(T)::count(object : T) : Integer	Возвращает количество object в целевой коллекции
X(T)::includes(object : T) : Boolean	Возвращает true, если целевая коллекция содержит object

Таблица 25.11 (продолжение)

Операции запроса Операция над коллекциями	Семантика
X(T)::excludes(object : T) : Boolean	Возвращает true, если целевая коллекция *не* содержит object
X(T)::includesAll(c : Collection(T)) : Boolean	Возвращает true, если в целевой коллекции содержатся все элементы c
X(T)::excludesAll(c : Collection(T)) : Boolean	Возвращает true, если в целевой коллекции *не* содержатся элементы c (если хотя бы один элемент c содержится в целевой коллекции, результат – false).
X(T)::isEmpty() : Boolean	Возвращает true, если целевая коллекция пуста, в противном случае возвращает false
X(T)::notEmpty() : Boolean	Возвращает true, если целевая коллекция не пуста, в противном случае возвращает false

25.8.6.5. Операции доступа

Только упорядоченные коллекции OrderedSet и Sequence обеспечивают возможность прямого доступа к своим элементам по их положению в коллекции (табл. 25.12). Чтобы организовать доступ к элементам неупорядоченных коллекций, необходимо перебрать все элементы коллекции от начала до конца.

Таблица 25.12

Операции доступа Операция над коллекциями	Семантика
OrderedSet(T)::first() : T Sequence(T)::first() : T	Возвращает первый элемент коллекции
OrderedSet(T)::last() : T Sequence(T)::last() : T	Возвращает последний элемент коллекции
OrderedSet::at(i) : T Sequence::at(i) : T	Возвращает элемент в позиции i
OrderedSet::indexOf(T) : Integer	Возвращает индекс объекта-параметра в OrderedSet

25.8.6.6. Операции выбора

Операции выбора (табл. 25.13) возвращают новые коллекции, являющиеся подмножествами или надмножествами целевой коллекции. Мы воспользовались диаграммами Венна (Venn) для иллюстрации операций с множествами: объединения, пересечения, строгой дизъюнкции и дополнения.

Таблица 25.13

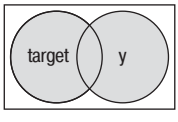
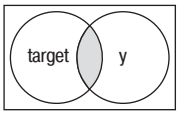
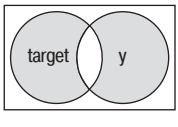
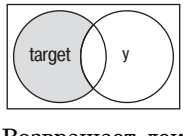
Операции выбора **Операция над коллекциями**	**Семантика**
X(T)::union(y : X(T)) : X(T)	Возвращает новую коллекцию, которая является результатом присоединения y к целевой коллекции; тип новой коллекции всегда соответствует типу целевой коллекции. Дублирующиеся элементы удаляются, порядок устанавливается в случае необходимости. 
Set(T)::intersection(y : Set(T)) : Set(T) OrderedSet(T)::intersection(y : OrderedSet(T)) : OrderedSet(T)	Возвращает новую коллекцию, содержащую элементы, присутствующие и в y, и в целевой коллекции. 
Set(T)::symmetricDifference(y : Set(T)) : Set(T) OrderedSet(T)::symmetricDifference(y : OrderedSet(T)) : OrderedSet(T)	Возвращает новый Set, содержащий элементы, которые существуют в целевой коллекции и в y, но не в обеих. 
Set(T)::-(y : Set(T)) : Set(T) OrderedSet(T)::-(y : OrderedSet(T)) : OrderedSet(T)	Возвращает новый Set, содержащий все элементы целевой коллекции, которых *нет* в y. В теории множеств возвращаемое множество является *дополнением* a относительно b. 
X(T)::product(y : X(T2)) : Set(Tuple(first : T, second : T2))	Возвращает декартово произведение целевой коллекции и y – это Set объектов Tuple{ first=a,second=b }, где a – член целевой коллекции, a b – член y например Set{ 'a',' b' }->product(Set { '1','2' })

Таблица 25.13 (продолжение)

Операции выбора Операция над коллекциями	Семантика
	возвращает Set{ Tuple{ first='a', second='1' }, Tuple{ first='a',second='2' }, Tuple{ first='b', second='1' }, Tuple{ first='b',second='2' } }
X(T)::including(object : T) : X(T)	Возвращает новую коллекцию, включающую содержимое целевой коллекции плюс object. Если коллекция упорядочена, object добавляется в ее конец.
X(T)::excluding(object : T) : X(T)	Возвращает новую коллекцию, в которой удалены все вхождения object.
Sequence(T)::subSequence(i : Integer, j : Integer) : Sequence(T)	Возвращает новый Sequence, в который входят элементы целевой коллекции, начиная от элемента с индексом i, заканчивая элементом с индексом j.
OrderedSet::subOrderedSet (i : Integer, j : Integer) : OrderedSet(T)	Возвращает новый OrderedSet, в который входят элементы целевого OrderedSet, начиная с элемента с индексом i, заканчивая элементом с индексом j.
OrderedSet(T)::append(object : T) : OrderedSet(T) Sequence(T)::append(object : T) : Sequence(T)	Возвращает новую коллекцию с object, добавленным в конце.
OrderedSet(T)::prepend(object : T) : OrderedSet(T) Sequence(T)::prepend(object : T) : Sequence(T)	Возвращает новую коллекцию с object, добавленным в начале.
OrderedSet(T)::insertAt(index : Integer, object : T) : OrderedSet(T) Sequence(T)::insertAt(index : Integer, object : T) : Sequence(T)	Возвращает новую коллекцию с object, вставленным в позицию по указанному индексу.

25.8.7. Итерационные операции

Итерационные операции позволяют циклически перебирать элементы коллекции. Их общая форма показана на рис. 25.7.

Слова в угловых скобках (<...>) должны быть заменены соответствующими величинами. Слова, выделенные серым цветом, обозначают необязательные части.

```
коллекция-><операцияИтератора>( переменнаяИтератора> : <Тип> |
                                    < выражениеИтератора>
                              )
```

Рис. 25.7. *Общая форма итерационных операций*

Операции итератора работают следующим образом: операцияИтератора поочередно посещает каждый элемент коллекции. Текущий элемент представлен переменнойИтератора. Выражение выражениеИтератора применяется к переменнойИтератора для получения результата. Каждая операцияИтератора обрабатывает результат по-своему.

Указывать Тип переменной итератора необязательно, поскольку он всегда аналогичен типу элементов коллекции. Сама переменнаяИтератора является необязательной. Когда выбран любой из элементов коллекции, все его возможности *автоматически* доступны выражениюИтератора. Доступ к ним может осуществляться напрямую по имени. Например, если элемент является объектом *BankAccount*, имеющим атрибут balance, выражениеИтератора может ссылаться непосредственно на balance.

Однако отсутствие в записи переменнойИтератора может быть чревато неприятностями. Мы считаем это плохим стилем, поскольку выражениеИтератора сначала ищет все необходимые ему переменные в своем пространстве имен, а если не может найти переменную, ищет в окружающих пространствах имен. Если переменнаяИтератора пропущена, существует опасность, что выражениеИтератора обнаружит не то, что надо.

Мы разделили операции итератора на логические операции (возвращающие значение типа Boolean) и операции выбора (возвращающие выборку из коллекции). Все операции представлены в табл. 25.14.

Таблица 25.14

Логические операции итератора	Семантика
X(T)::exists(i : T | iteratorExpression) : Boolean	Возвращает true, если выражение итератора (iteratorExpression) принимает значение true, по крайней мере, для одного значения i, в противном случае возвращает false.
X(T)::forAll(i : T | iteratorExpression) : Boolean	Возвращает true, если iteratorExpression принимает значение true для всех значений i, в противном случае возвращает false.
X(T)::forAll(i : T, j : T ..., n : T | iteratorExpression) : Boolean	Возвращает true, если iteratorExpression принимает значение true для каждого { i, j ... n } Tuple, в противном случае возвращает false. Множество пар { i, j ... n } – это декартово произведение целевой коллекции с самой собой.

Таблица 25.14 (продолжение)

Логические операции итератора	Семантика
X(T)::isUnique(i : T | iteratorExpression) : Boolean	Возвращает true, если iteratorExpression имеет уникальное значение для каждого значения i, в противном случае возвращает false.
X(T)::one(i : T | iteratorExpression) : Boolean	Возвращает true, если iteratorExpression принимает значение true только для одного значения i, в противном случае возвращает false.
Операции выбора для итератора	**Семантика**
X(T)::any(i : T | iteratorExpression) : T	Возвращает случайный элемент целевой коллекции, для которого iteratorExpression имеет значение true.
X(T)::collect(i : T | iteratorExpression) : Bag(T)	Возвращает Bag, содержащий результаты однократного выполнения iteratorExpression для каждого элемента целевой коллекции (нотацию краткой записи collect(...) см. в разделе 25.9.2).
X(T)::collectNested(i : T | iteratorExpression) : Bag(T)	Возвращает Bag коллекций, содержащий результаты однократного выполнения iteratorExpression для каждого элемента целевой коллекции. Поддерживает вложение целевой коллекции в результирующую коллекцию.
X(T)::select(i : T | iteratorExpression) : X(T)	Возвращает коллекцию, содержащую элементы целевой коллекции, для которых iteratorExpression принимает значение true.
X(T)::reject(i : T | iteratorExpression) : X(T)	Возвращает коллекцию, содержащую элементы целевой коллекции, для которых iteratorExpression принимает значение false.
X(T)::sortedBy(i : T | iteratorExpression) : X(T)	Возвращает коллекцию, содержащую элементы целевой коллекции, упорядоченные соответственно iteratorExpression. Тип переменной итератора (iteratorVariable) *должен* быть типом, в котором определен оператор <.

Стоит поближе рассмотреть forAll(...). Данная операция имеет две формы. Первая форма – с одной переменной итератора (iteratorVariable), вторая – с множеством. Вторая форма является сокращенной записью нескольких вложенных операций forAll(...).

Например, рассмотрим две вложенные операции forAll(...):

```
c->forAll( i | c->forAll( j | iteratorExpression ) )
```

Они могут быть записаны следующим образом:

```
c->forAll( i, j | iteratorExpression )
```

Результатом применения обеих форм записи операции является перебор множества пар { i, j }, т. е. декартово умножение c на саму себя. Проясним это с помощью примера. Пусть

```
c = Set{ x, y, z }
```

Декартовым произведением c с самой собой является Set

```
{ {x,x}, {x,y}, {x,z}, {y,x}, {y,y}, {y,z}, {z,x}, {z,y}, {z,z} }
```

Затем c->forAll(i, j | iteratorExpression) осуществляет перебор каждого подмножества этого Set, присваивая каждому i и j по одному элементу подмножества. После этого i и j можно использовать в выражении итератора.

Мы полагаем, что форма forAll(...) с несколькими параметрами может сбить с толку, и ее применения следует избегать.

Все эти итерационные операции (кроме forAll(...) с несколькими параметрами) являются особыми случаями более общей операции iterate, которая рассматривается в следующем разделе.

25.8.7.1. Операция iterate

С помощью OCL-операции iterate можно создавать собственные специальные итерации. Ее форма представлена на рис. 25.8.

Как видим, кроме переменнойИтератора и ее Типа (который в данном случае является *обязательным*) здесь присутствует результирующаяПеременная, тип которой может отличаться от типа переменнойИтератора. результирующаяПеременная получает начальное значение от выраженияИнициализации, а ее конечное значение формируется в результате последовательных применений выраженияИтератора.

Принцип работы операции iterate следующий. выражениеИнициализации инициализирует результирующуюПеременную некоторым значением. Затем операция iterate выполняет выражениеИтератора для каждого члена коллекции по очереди, используя переменнуюИтератора и текущее значение результирующейПеременной. Результатом вычисления выраженияИтератора становится новое значение результирующейПеременной, которое используется при выполнении выраженияИтератора для следующего элемента коллекции. Значение операции iterate(...) – это конечное значение результирующейПеременной.

```
коллекция -> iterate( <переменнаяИтератора> : <Тип>
                      <результирующаяПеременная> : <РезультирующийТип> = <выражениеИнициализации> |
                          <выражениеИтератора>
                     )
```

Рис. 25.8. *Форма операции iterate*

Вы легко усвоите принцип действия операции iterate из следующего примера:

```
Bag{ 1, 2, 3, 4, 5 }->iterate( number : Integer;
                               sum : Integer = 0 |
                               sum + number
                             )
```

В результате этого выражения получаем сумму чисел в Bag. В данном случае она равняется 15. Это абсолютно эквивалентно следующей операции:

```
Bag{ 1, 2, 3, 4, 5 }->sum()
```

Операция iterate – самый универсальный итератор. Она может использоваться для моделирования всех остальных итераторов. Ниже приведен пример, который выбирает все положительные числа множества (Set).

```
Set{ -2, -3, 1, 2 }->iterate( number : Integer;
        positiveNumbers : Set(Integer) = Set{} |     -- создаем пустой Set
        if number >= 0 then                          -- пропускаем отрицательные числа
            positiveNumbers->including( number )  -- добавляем число в конец
                                                     -- результирующего Set
        else
            positiveNumbers                       -- просто возвращаем resultVariable
        endif
)
```

Это абсолютно эквивалентно следующему:

```
Set{ -2, -3, 1, 2 }->select( number : Integer | number >= 0 )
```

25.9. Навигация в OCL

> Навигация – это способность перемещаться от исходного объекта к одному или более целевым объектам.

Навигация – это процесс, при помощи которого можно проследить связи от исходного объекта к одному или более целевым объектам.

Навигация – вероятно, самый сложный и запутанный вопрос OCL. До сих пор для написания OCL-выражения необходимо было знать путь от контекста выражения к другим элементам модели, на которые надо сослаться. Это означает, что OCL должен использоваться как навигационный язык.

Навигационные выражения OCL могут ссылаться на любое из следующего:

- классификаторы;

- атрибуты;
- концы ассоциаций;
- операции запроса (операции, свойство isQuery которых имеет значение true).

В спецификации OCL [OCL1] их называют *свойствами*.

В следующем разделе рассматривается простая навигация в рамках экземпляра контекста, а далее – навигация по отношениям с кратностью 1 и больше 1.

25.9.1. Навигация в рамках экземпляра контекста

Давайте рассмотрим простой пример навигации для доступа к возможностям экземпляра контекста. На рис. 25.9 показан класс A с единственным атрибутом a1 и единственной операцией op1().

Предполагая, что класс A является контекстом выражения, можно написать навигационные выражения OCL, перечисленные в табл. 25.15.

Таблица 25.15

Навигационное выражение	Семантика
self	Экземпляр контекста – экземпляр класса A
self.a1 a1	Значение атрибута a1 экземпляра контекста
self.op1() op1()	Результат вызова op1() экземпляра контекста Операция op1() *должна* быть операцией запроса

Здесь необходимо отметить несколько важных моментов.

- Доступ к экземпляру контекста осуществляется с помощью ключевого слова self.
- Доступ к свойствам экземпляра контекста осуществляется напрямую или с помощью ключевого слова self и оператора «точка». Придерживаясь хорошего стиля, мы предпочитаем явно использовать self и оператор «точка».
- Единственные операции, к которым можно организовать доступ, – операции запроса.

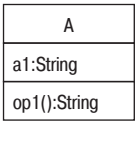


***Рис. 25.9.** Класс A с одним атрибутом и одной операцией*

25.9.2. Навигация по ассоциациям

Навигация по ассоциациям немного сложнее. Обычно навигацию можно осуществлять только по ассоциациям, допускающим навигацию, и можно получать доступ только к открытым возможностям классов. Однако спецификация OCL позволяет интерпретатору OCL иметь *необязательную* возможность проходить не допускающие навигацию ассоциации и получать доступ к закрытым и защищенным свойствам. Возможности используемого для обработки OCL-выражения интерпретатора OCL необходимо сверять по спецификации.

На рис. 25.10 представлены некоторые выражения навигации по ассоциации между двумя классами A и B, где кратность на конце b равна 1.

> Для навигации по ассоциациям необходимо использовать оператор «точка».

Навигация по ассоциации осуществляется с помощью оператора «точка» так, как если бы имя роли являлось атрибутом контекстного класса. Выражение навигации может возвращать объект (или объекты), находящийся на целевом конце, значения его атрибутов и результаты его операций.

> Семантика навигации зависит от кратности на целевом конце ассоциации.

Навигация усложняется, когда кратность на целевом конце ассоциации больше 1, потому что семантика навигации зависит от кратности.

На рис. 25.11 показаны некоторые выражения для навигации по ассоциации между двумя классами C и D, где кратность на конце d – много.

Навигационное выражение

```
self.d
```

возвращает Set(D) объектов d.

Это означает, что оператор «точка» перегружен. Когда кратность на целевом конце равна 1 или 0..1, он возвращает объект того же типа, что

Пример модели			Навигационные выражения (A - контекст выражения)	
			Выражение	Значение
A	b	B	self	Экземпляр контекста – экземпляр A
a1:String	1	b1:String	self.b	Объект типа B
контекст		op1():String	self.b.b1	Значение атрибута B::b1
			self.b.op1()	Результат операции B::op1()

Рис. 25.10. *Выражения для навигации по ассоциации между класами A и B с кратностью на конце, равной 1*

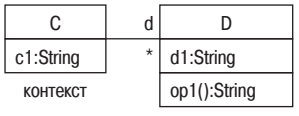
Пример модели	Навигационные выражения	
	Выражение	Значение
	self	Экземпляр контекста – экземпляр C
	self.d	Set(D) объектов типа D
	self.d.d1	Bag(String) значений атрибута D::d1 Краткая запись для self.d->collect(d1)
	self.d.op1()	Bag(String) результатов операции D::op1() Краткая запись для self.d->collect(op1())

Рис. 25.11. *Выражения для навигации по ассоциации между класами A и B с кратностью на конце – * (много)*

и целевой класс. Когда кратность больше 1, он возвращает Set объектов целевого класса.

> По умолчанию, если кратность >1, оператор «точка» возвращает Set.

По умолчанию, если кратность – «много», оператор «точка» возвращает Set объектов. Однако с помощью свойств ассоциаций, приведенных в табл. 25.16, можно задавать тип возвращаемой коллекции.

Таблица 25.16

OCL-коллекция	Свойства конца ассоциации
Set	{ unordered, unique } – применяется по умолчанию
OrderedSet	{ ordered, unique }
Bag	{ unordered, nonunique }
Sequence	{ ordered, nonunique }

> Доступ к свойству коллекции является сокращенной записью операции collect(...).

Когда осуществляется доступ к свойству коллекции, например

```
self.d.d1
```

это выражение является сокращенной записью для

```
self.d->collect( d1 )
```

Возможно, вы помните из раздела 25.8.7, что collect(iteratorExpression) возвращает Bag, содержащий результаты выполнения iteratorExpression для каждого элемента коллекции. В данном случае возвращается Bag значений атрибута d1 для каждого объекта D в Set(D), полученном путем обхода self.d.

Аналогично

```
self.d.op1()
```

является сокращенной записью для

```
self.d->collect( d.op1() )
```

Результатом этого выражения является Bag, содержащий возвращаемые значения операции op1(), примененной к каждому объекту D в Set(D), полученном путем обхода self.d.

Операция collect() всегда возвращает плоскую коллекцию. Если необходимо сохранить вложенность целевой коллекции в возвращаемой коллекции, следует использовать операцию collectNested().

25.9.3. Навигация по нескольким ассоциациям

В данном разделе рассматривается навигация по нескольким ассоциациям.

В принципе можно осуществлять навигацию по любому числу ассоциаций. На практике навигация ограничивается максимум двумя ассоциациями, потому что пространные навигационные выражения чреваты ошибками и могут быть сложны для понимания. Они также делают OCL-выражения слишком многословными.

Давайте рассмотрим простой пример навигации по двум ассоциациям (рис. 25.12).

> Результатом навигации по ассоциации с кратностью >1 является Bag.

Можно заметить, что результатом навигации по ассоциации с кратностью больше 1 всегда является Bag, потому что эта операция эквивалентна применению collect(...). Например, выражение

```
self.k.l.l1
```

эквивалентно

```
self.k->collect( l )->collect( l1 )
```

Аналогичным образом можно провести навигацию по большему числу ассоциаций, но этого делать не рекомендуется.

25.10. Подробно о типах OCL-выражений

В разделе 25.7 были представлены различные типы OCL-выражений. Теперь, рассмотрев синтаксис OCL, мы можем подробно остановиться на каждом из этих типов. В качестве примера воспользуемся простой моделью, приведенной на рис. 25.13.

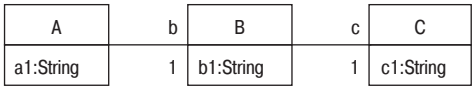
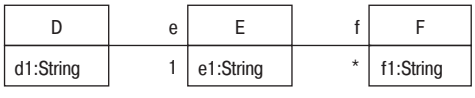
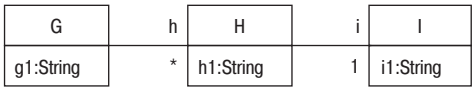
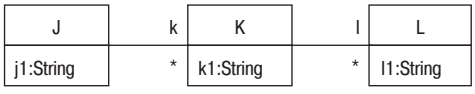
Пример модели	Навигационные выражения	
	Выражение	Значение
	self self.b self.b.b1 self.b.c self.b.c.c1	Экземпляр контекста - экземпляр A Объект типа B Значение атрибута B::b1 Объект типа C Значение атрибута C::c1
	self self.e self.e.e1 self.e.f self.e.f.f1	Экземпляр контекста - экземпляр D Объект типа E Значение атрибута E::e1 Set(F) объектов типа F Bag(String) значений атрибута F::f1
	self self.h self.h.h1 self.h.i self.h.i.i1	Экземпляр контекста - экземпляр G Set(H) объектов типа H Bag(String) значений атрибута H::h1 Bag(I) объектов типа I Bag(String) значений атрибута I::i1
	self self.k self.k.k1 self.k.l self.k.l.l1	Экземпляр контекста - экземпляр J Set(K) объектов типа K Bag(String) значений атрибута K::k1 Bag(L) объектов типа L Bag(String) значений атрибута L::l1

Рис. 25.12. *Навигация по двум ассоциациям*

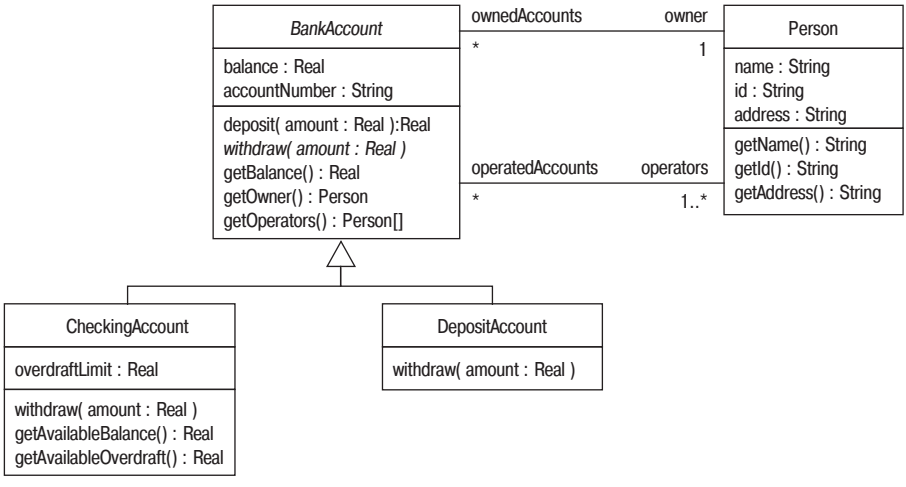


Рис. 25.13. *Модель банковского счета*

25.10.1. inv:

Инвариант – это нечто, что должно быть истинным для всех экземпляров контекстного классификатора.

Рассмотрим модель простого банковского счета (рис. 25.13). Классы CheckingAccount и DepositAccount подчиняются четырем бизнес-правилам.

1. Ни на одном из счетов кредит не может быть превышен более чем на $1000.
2. CheckingAccount предоставляет возможность превышения кредита. Со счета может быть снята сумма, не превышающая установленного ограничения по превышению кредита.
3. На счетах DepositAccount не допускается превышение кредита.
4. Каждый accountNumber должен быть уникальным.

Первое правило, состоящее в том, что ни на одном из счетов кредит не может быть превышен более чем на $1000, можно представить как инвариант класса *BankAccount*, потому что оно должно выполняться для всех экземпляров *BankAccount* (т. е. всех экземпляров его подклассов).

```
context BankAccount
  inv balanceValue:
    -- баланс BankAccount должен быть ->1000.0
    self.balance >= (-1000.0)
```

Этот инвариант наследуется двумя подклассами: CheckingAccount и DepositAccount. Для обеспечения сохранения принципа замещаемости (раздел 10.2) подклассы могут *усилить* инвариант, но ни в коем случае не *ослабить* его.

> Подкласс может усилить инвариант, но не ослабить его.

Правила 1 и 2 могут быть выражены как инварианты класса CheckingAccount:

```
context CheckingAccount
  inv balanceValue:
    -- превышение кредита на CheckingAccount не должно быть
    -- больше установленного ограничения
    self.balance >= (-overdraftLimit)
  inv maximumOverdraftLimit:
    -- превышение кредита на CheckingAccount не должно быть больше 1000.0
    self.overdraftLimit <= 1000.0
```

Правило 3 может быть выражено как инвариант класса DepositAccount:

```
context DepositAccount
  inv balanceValue:
    -- баланс DepositAccount должен быть нулевым или положительным
    self.balance >= 0.0
```

Обратите внимание, как оба этих класса, переопределив инвариант класса BankAccount::balance, усилили его.

Ограничение, касающееся уникальности accountNumber каждого счета, можно представить как инвариант класса *BankAccount*:

```
context BankAccount
  inv uniqueAccountNumber:
    -- у каждого BankAccount должен быть уникальный accountNumber
    BankAccount::allInstances( )->isUnique( account | account.accountNumber )
```

Из рис. 25.13 видно, что у каждого *BankAccount* есть единственный владелец (owner) и один или более операторов (operator). Владелец – это человек (Person), которому принадлежит счет, а операторы – это люди (People), имеющие право снимать деньги и имеющие доступ к информации счета. Существует бизнес-ограничение о том, что owner должен быть также и operator. Данное ограничение может быть отражено следующим образом:

```
context BankAccount
  inv ownerIsOperator:
    -- владельцем BankAccount должен быть один из его операторов
    self.operators->includes( self.owner )
```

Для Person можно записать такое ограничение:

```
context Person
  inv ownedAccountsSubsetOfOperatedAccounts:
    -- счета, принадлежащие Person (ownedAccount), должны быть
    -- подмножеством счетов, управляемых этим Person (operatedAccount)
    self.operatedAccounts->includesAll( self.ownedAccounts )
```

При сравнении объектов в OCL-выражениях необходимо помнить, что они могут быть:

- идентичные – каждый объект ссылается на одну и ту же область памяти (имеют идентичные объектные ссылки);
- эквивалентные – каждый объект имеет одинаковый набор значений атрибутов, но разные объектные ссылки.

В приведенных выше OCL-выражениях мы всегда аккуратны при сравнении объектов на основании их идентичности или эквивалентности соответственно. С этим надо проявлять осторожность. Например, сравнение объектов *BankAccount* (сравнение на основании идентичности) не то же самое, что сравнение accountNumber этих объектов (сравнение на основании эквивалентности).

25.10.2. pre:, post: и @pre

Предусловия и постусловия применяются к операциям. Их экземпляром контекста является экземпляр классификатора, которому принадлежат эти операции.

pre: и post: применяются к операциям.

- Предусловия определяют сущности, которые должны быть истинны *перед* выполнением операции.
- Постусловия определяют сущности, которые должны быть истинны *после* выполнения операции.

Вернемся к нашему примеру *BankAccount* на рис. 25.13 и рассмотрим операцию deposit(...), унаследованную обоими подклассами, CheckingAccount и DepositAccount, от *BankAccount*. Установлено два бизнес-правила.

1. Сумма (amount) вклада должна быть больше нуля.
2. После выполнения операции сумма должна быть добавлена в баланс (balance).

Эти правила могут быть выражены кратко и точно в виде предусловий и постусловий операции BankAccount::deposit(...):

```
context BankAccount::deposit( amount : Real ) : Real
  pre amountToDepositGreaterThanZero:
    -- сумма вклада должна быть больше нуля
    amount >0

  post depositSucceeded:
    -- окончательный баланс должен быть суммой исходного баланса и вклада
    self.balance = self.balance@pre + amount
```

Предусловие amountToDepositGreaterThanZero (сумма вклада больше нуля) должно быть истинным, чтобы обеспечить возможность выполнения операции. Оно гарантирует:

- невозможность нулевых вкладов;
- невозможность вкладов с отрицательным значением суммы.

Постусловие depositSucceeded (вклад сделан успешно) должно быть true после выполнения операции. Оно определяет увеличение исходного баланса (balance@pre) на сумму вклада (amount) для получения окончательного баланса.

attributeName@pre ссылается на значение перед выполнением операции.

Обратите внимание на ключевое слово @pre. Оно может использоваться *только* в постусловиях. Атрибут balance принимает одно значение до выполнения операции и другое значение после ее выполнения. Выражение balance@pre ссылается на значение balance *перед* выполнением операции. Часто в постусловии необходимо сослаться на исходное значение чего-либо.

Для полноты информации приводим ограничения, налагаемые на операцию BankAccount::withdraw(...).

```
context BankAccount::withdraw( amount : Real )
  pre amountToWithdrawGreaterThanZero:
    -- снимаемая сумма должна быть больше нуля
    amount >0

  post withdrawalSucceeded:
    -- окончательный баланс – это разность исходного баланса и снятой суммы
    self.balance = self.balance@pre-amount
```

Прежде чем завершить обсуждение предусловий и постусловий, необходимо рассмотреть наследование. Когда подкласс переопределяет операцию, он принимает предусловия и постусловия переопределяемой операции. Он может только изменить их.

- Переопределенная операция может только *ослабить* предусловие.
- Переопределенная операция может только *усилить* постусловие.

Эти ограничения гарантируют сохранение принципа замещаемости (раздел 10.2).

25.10.3. body:

OCL можно использовать для определения результата операции запроса. Все операции getXXX() в нашей простой модели BankAccount (рис. 25.13) являются операциями запроса.

```
BankAccount::getBalance( ) : Real
BankAccount::getOwner() : Person
BankAccount::getOperators() : Set( Person )
CheckingAccount::getAvailableBalance() : Real
CheckingAccount::getAvailableOverdraft() : Real
```

OCL-выражения для операций запроса *BankAccount* тривиальны, и обычно их написание не должно вызывать затруднений. Выражения приведены ниже в качестве примера:

```
context BankAccount::getBalance( ) : Real
  body:
    self.balance

context BankAccount::getOwner( ) : Person
  body:
    self.owner

context BankAccount::getOperators() : Set(Person)
  body:
    self.operators
```

Операции запроса класса CheckingAccount более интересны:

```
context CheckingAccount::getAvailableBalance() : Real
  body:
    -- можно снимать сумму, не превышающую ограничения по превышению кредита
    self.balance + self.overdraftLimit
```

```
context CheckingAccount::getAvailableOverdraft() : Real
  body:
   if self.balance >= 0 then
     -- возможность превышения кредита доступна полностью
     self.overdraftLimit
   else
     -- возможность превышения кредита использована частично
     self.balance + self.overdraftLimit
   endif
```

Как видите, в этих двух операциях запроса OCL определяет, как вычисляется результат операции. Возвращаемое значение операции – это результат вычисления OCL-выражения.

25.10.4. init:

OCL может использоваться для задания начального значения атрибутов. Например:

```
context BankAccount::balance
init:
  0
```

Обычно эта возможность OCL используется в случае сложности инициализации. Простые инициализации (как та, что приведена выше) лучше всего размещать прямо в ячейке атрибутов класса.

25.10.5. def:

OCL позволяет добавлять атрибуты и операции в классификатор с помощью стереотипа «OclHelper». Они могут использоваться только в OCL-выражениях. Добавленные атрибуты в OCL называют переменными. Их применение во многом аналогично использованию переменных в других языках программирования. Добавленные операции называются вспомогательными операциями, потому что они «помогают» в OCL-выражениях.

> def: позволяет определить переменные и вспомогательные операции классификатора для использования в других OCL-выражениях.

Переменные и вспомогательные операции используются для упрощения OCL-выражений.

Рассмотрим пример. Возьмем ограничения, определенные ранее:

```
context CheckingAccount::getAvailableBalance() : Real
  body:
     -- можно снимать сумму, не превышающую ограничения превышения кредита
     balance + overdraftLimit

context CheckingAccount::getAvailableOverdraft() : Real
```

```
  body:
    if balance >= 0 then
      -- возможность превышения кредита доступна полностью
      overdraftLimit
    else
      -- возможность превышения кредита использована частично
      balance + overdraftLimit
    endif
```

Можно заметить, что операция balance + overdraftLimit присутствует в двух выражениях. Поэтому есть смысл определить ее один раз как переменную availableOverdraft (доступное превышение кредита), которая может использоваться обоими выражениями. Для этого применяется оператор def:.

```
context CheckingAccount
  def:
    availableBalance = balance + overdraftLimit
```

Теперь можно переписать эти два ограничения, используя введенную переменную:

```
context CheckingAccount::getAvailableBalance() : Real
  body:
    -- можно снимать сумму, не превышающую ограничения превышения кредита
    availableBalance

context CheckingAccount::getAvailableOverdraft() : Real
  body:
    if balance >= 0 then
      -- возможность превышения кредита доступна полностью
      overdraftLimit
    else
      -- возможность превышения кредита использована частично
      availableBalance
    endif
```

Можно также определить вспомогательные операции. Например, в OCL-выражениях может быть полезной операция для проверки возможности снятия денег со счета. Ее можно было бы определить следующим образом:

```
context CheckingAccount
 def:
   canWithdraw( amount : Real ) : Boolean = ( (availableBalance -amount ) >= 0 )
```

25.10.6. Выражения let

Если def: предоставляет возможность определять переменные уровня контекста выражения, то let позволяет задавать переменные, область действия которых ограничена конкретным OCL-выражением. Эти пере-

менные подобны локальным переменным в обычных языках программирования. И назначение их практически такое же – хранение вычисленного значения, используемого в выражении более одного раза.

> let определяет локальную переменную OCL-выражения.

Выражение let состоит из двух частей – let и in.

```
let <имяПеременной>:<типПеременной> = <выражениеLet> in
  <использующееВыражение>
```

В первой части значение выражения let (<выражениеLet>) присваивается переменной (<имяПеременной>). Вторая часть определяет OCL-выражение, являющееся областью действия переменной и местом, где она может использоваться (<использующееВыражение>).

В нашем примере банковского счета на самом деле нет *необходимости* в выражении let. Однако для иллюстрации рассмотрим пример, в котором определена переменная originalBalance (исходный баланс), локальная для ограничения withdrawalSucceeded.

```
context BankAccount::withdraw( amount : Real )
  post withdrawalSucceeded:
    let originalBalance : Real = self.balance@pre in
      -- окончательный баланс – это исходный баланс минус снятая сумма
      self.balance = originalBalance -amount
```

25.10.7. derive:

OCL может использоваться для определения значений производных атрибутов.

> Значения производных атрибутов определяются с помощью derive.

Наш пример с банковским счетом можно реорганизовать и выразить переменную баланса и переменную превышения кредита с помощью производных атрибутов (рис. 25.14).

Правила вывода этих производных атрибутов можно описать следующим образом:

```
context CheckingAccount::availableBalance : Real
  derive:
    -- можно снимать сумму, не превышающую ограничения по превышению кредита
    balance + overDraftLimit

context CheckingAccount::availableOverdraft : Real
  derive:
    if balance >= 0 then
      -- возможность превышения кредита доступна полностью
      overdraftLimit
```

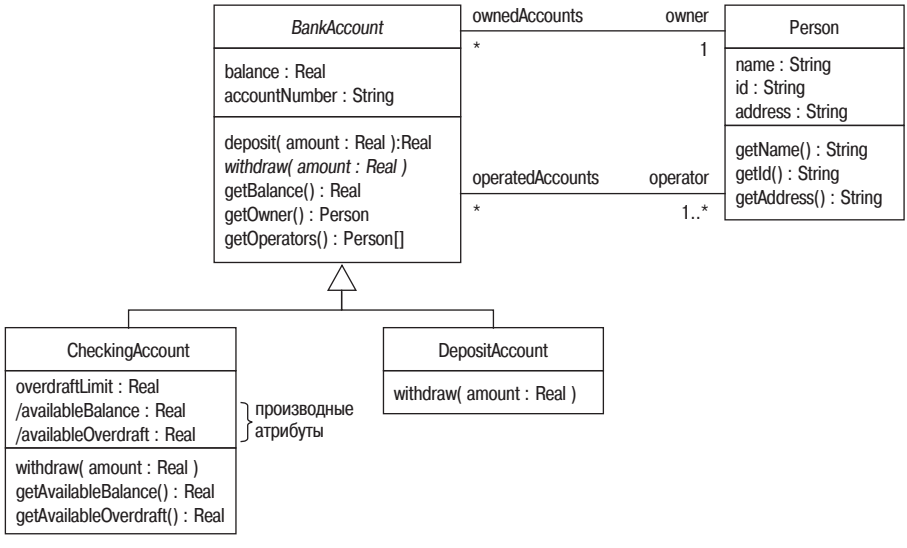


Рис. 25.14. *Производные атрибуты*

```
    else
      -- возможность превышения кредита использована частично
      overdraftLimit + balance
    endif
```

Это упрощает определение операций CheckingAccount::getAvailableBalance() и CheckingAccount::getAvailableOverdraft():

```
context CheckingAccount::getAvailableBalance() : Real
  body:
    availableBalance

context CheckingAccount::getAvailableOverdraft() : Real
  body:
    availableOverdraft
```

25.11. OCL на диаграммах других типов

До сих пор мы рассматривали применение OCL только на диаграммах классов. Однако OCL может также использоваться на диаграммах других типов:

- на диаграммах взаимодействий (глава 12);
- на диаграммах деятельности (главы 14 и 15);
- на диаграммах состояний (конечные автоматы) (главы 21 и 22).

В следующих разделах рассмотрим применение OCL на диаграммах этих типов.

25.11.1. OCL на диаграммах взаимодействий

На диаграммах взаимодействий OCL используется для представления ограничений. Необходимо помнить, что с помощью OCL *нельзя* описать поведение, поскольку данный язык не имеет побочных эффектов.

На диаграммах взаимодействий OCL может использоваться везде, где необходимо сделать следующее:

- определить сторожевое условие;
- определить селектор линии жизни (раздел 12.6);
- определить параметры сообщения.

Рассмотрим пример использования OCL на диаграммах последовательностей. На рис. 25.15 изображена диаграмма классов простой системы электронной почты.

Класс EmailAddress (электронный адрес) представляет адрес электронной почты. Например, адрес электронной почты *jim@umlandtheunifiedprocess.com* был бы представлен как объект класса EmailAddress, как показано на рис. 25.16. В данном случае операция EmailAddress:get-

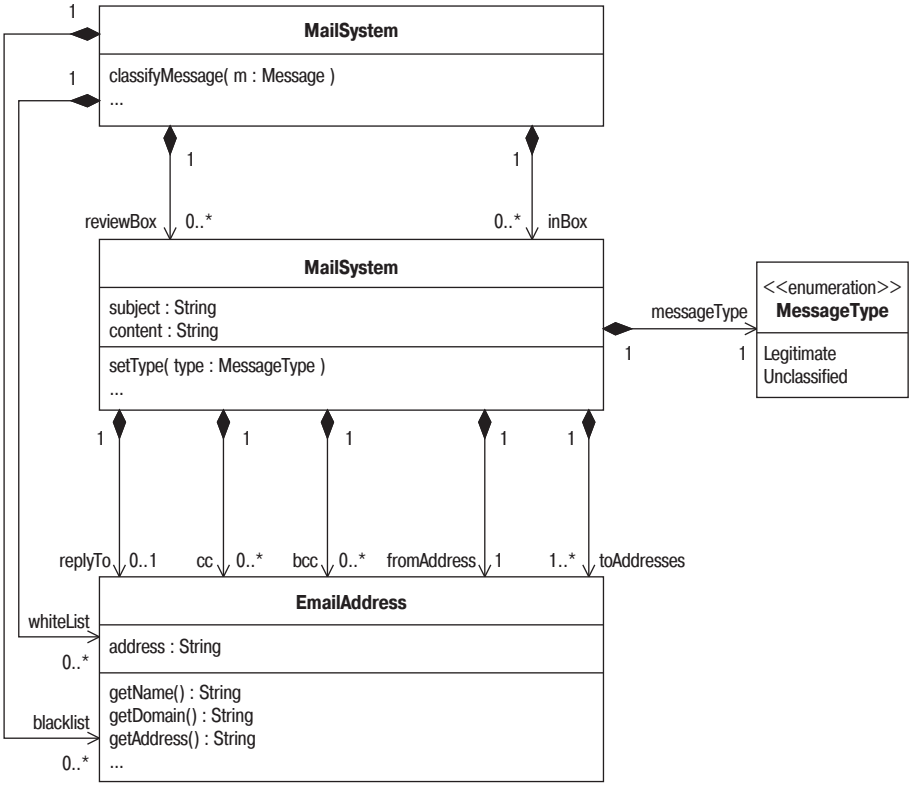


Рис. 25.15. *Диаграмма классов системы электронной почты*

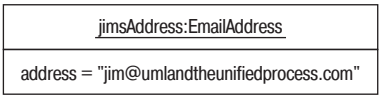


Рис. 25.16. *Адрес электронной почты jim@umlandtheunifiedprocess.com представлен как объект класса EmailAddress*

Name() возвращает "Jim", EmailAddress::getDomain() возвращает "umlandtheunifiedprocess.com" и EmailAddress::getAddress() возвращает "jim@umlandtheunifiedprocess.com". Объекты EmailAddress эквивалентны, если их атрибуты address (адрес) имеют одинаковое значение.

Система для обработки пришедшей почты пользуется политикой белый/черный список:

- Все почтовые сообщения, адрес отправителя (fromAddress) которых входит в черный список (blackList), удаляются.
- Все почтовые сообщения, fromAddress которых входит в белый список (whiteList), помещаются в ящик входящей почты (inBox).
- Все остальные почтовые сообщения помещаются в ящик для просмотра (reviewBox).
- Состояние пришедшего сообщения (Message) меняется соответственно его типу: спам (удаляется), Legitimate (допустимый) или Unclassified (неклассифицированный).

На рис. 25.17 показана диаграмма последовательностей для операции MailSystem::classifyMessage(m : Message). Диаграмма деятельностей для этой операции представлена в следующем разделе на рис. 25.18, и вы должны заметить, как эти диаграммы соответствуют друг другу. Диаграмма последовательностей определяет, какие классы и операции реализуют поведение, описанное диаграммой деятельностей.

MailSystem (система электронной почты) – контекст выражения. OCL использовался для определения условий в комбинированном фрагменте alt.

С помощью OCL также было задано состояние объектов на диаграмме (хотя это обычное применение OCL, поскольку OCL и UML имеют одинаковый синтаксис для описания состояний).

25.11.2. OCL на диаграммах деятельности

Диаграммы деятельности можно создавать для описания поведения любого элемента UML-моделирования. На диаграммах деятельностей OCL используется для определения:

- узлов вызова действия;
- сторожевых условий переходов;
- объектных узлов;
- состояния объекта.

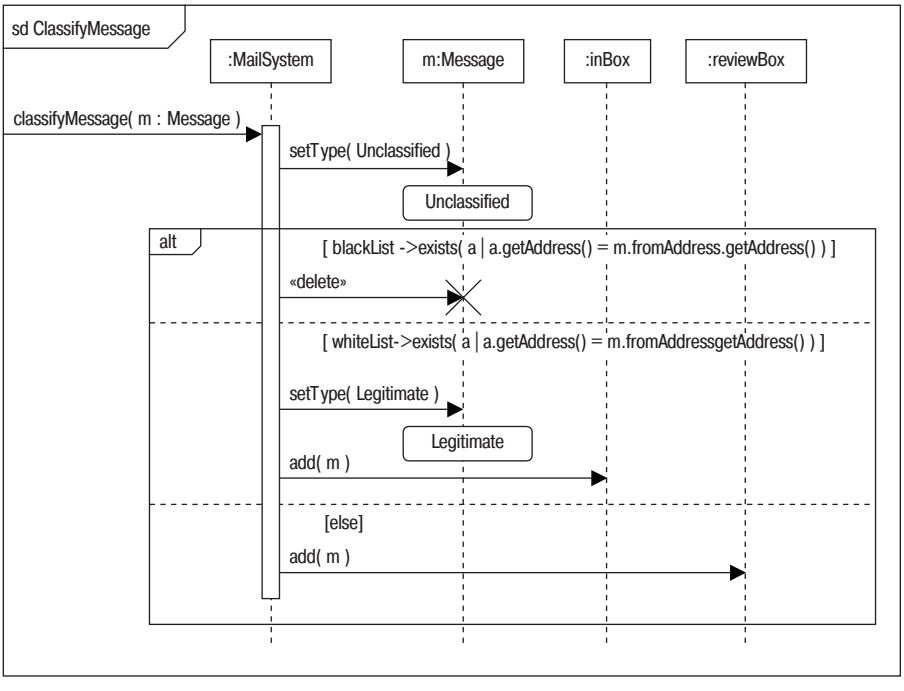


Рис. 25.17. *Диаграмма последовательностей для операции MailSystem::classifyMessage(m : Message)*

Например, на рис. 25.18 показана очень простая диаграмма деятельности для описания поведения ClassifyMailMessage (классифицировать почтовое сообщение) системы электронной почты, рассматриваемого в предыдущем разделе.

Как видите, OCL используется для описания объектов, их состояний и условий.

OCL применяется для точного моделирования, а диаграммы деятельности по своей природе не могут быть точными. Кроме того, они часто предназначаются для заказчиков, которые не являются специалистами в технической области. Поэтому целесообразность применения OCL на диаграммах этого типа довольно сомнительна. Например, ограничения на рис. 25.18 могли бы быть записаны просто на естественном языке (английском или русском). Рассматривая возможность использования OCL на диаграммах деятельности, всегда необходимо учитывать назначение диаграммы и ее целевую аудиторию.

25.11.3. OCL на диаграммах состояний

OCL используется в диаграммах состояний для определения:

- сторожевых условий;

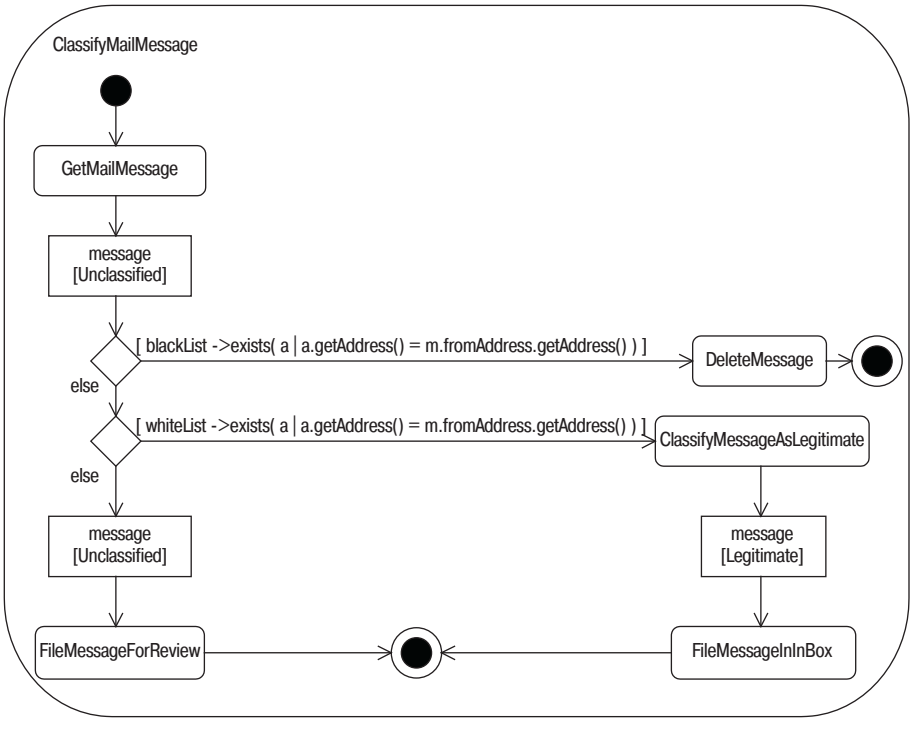


Рис. 25.18. *Диаграмма деятельностей*

- условий, налагаемых на состояния;
- целей действий;
- операций;
- значений параметров.

Экземпляр контекста – это экземпляр классификатора, которому принадлежит конечный автомат. В качестве примера рассмотрим конечный автомат CheckingAccountActive (активный текущий счет) класса CheckingAccount (рис. 25.19).

Из рисунка видно, что на диаграмме синтаксис OCL используется в сторожевых условиях. Чтобы конечный автомат был рабочим, необходимо учесть ограничения, которые записаны отдельно, чтобы не загромождать диаграмму.

```
context CheckingAccount::balance
  int:
    0

context CheckingAccount
  inv:
    oclInState( InCredit ) implies ( balance >= 0 )
```

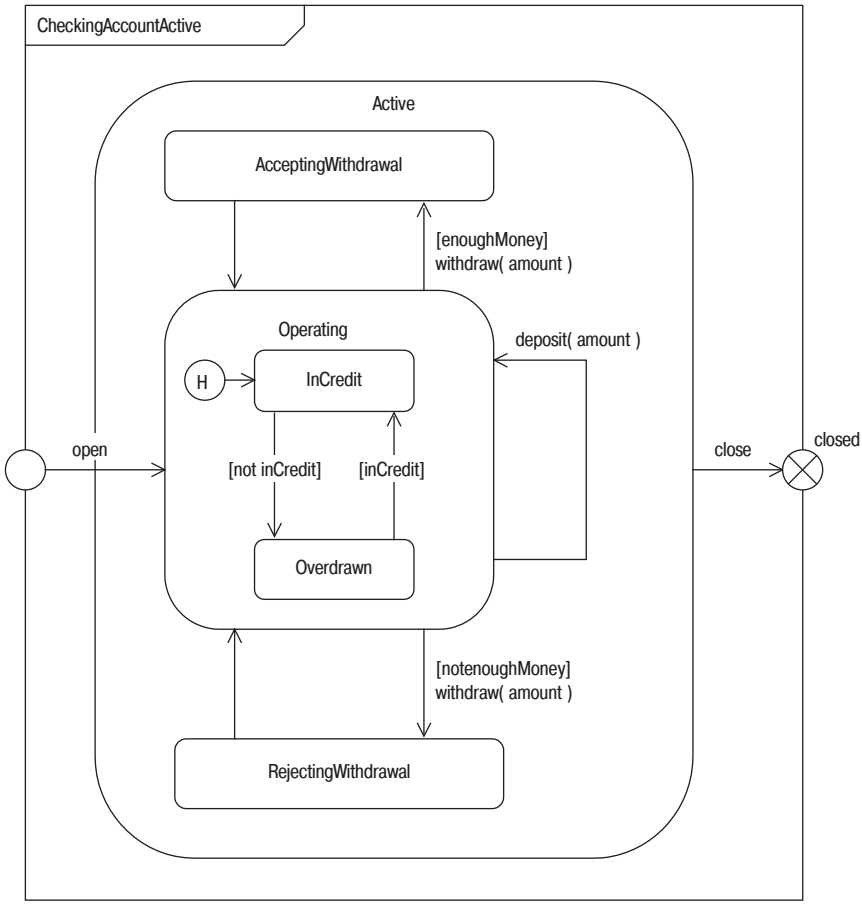


Рис. 25.19. *Конечный автомат CheckingAccountActive*

```
inv:
  oclInState( Overdrawn ) implies ( balance <0 )

def:
  availableBalance = ( balance + overdraftLimit )

def:
  enoughMoney = ( ( availableBalance – amount ) >= 0 )

def:
  inCredit = ( balance >= 0 )
```

Обратите внимание, как с помощью oclInState(InCredit) организована ссылка на состояние InCredit. В качестве альтернативы можно было бы прикрепить инвариант (balance >= 0) прямо к состоянию InCredit в виде примечания.

25.12. Дополнительные вопросы

В данном разделе рассматриваются некоторые не часто используемые аспекты OCL:

- навигация к и из классов-ассоциаций;
- навигация по квалифицированным ассоциациям;
- унаследованные ассоциации;
- OCLMessage.

25.12.1. Навигация к и от классов-ассоциаций

> Имя класса-ассоциации используется для навигации к классу-ассоциации.

Классы-ассоциации обсуждаются в разделе 9.4.5. Осуществить навигацию *к* классу-ассоциации можно с помощью его имени. Для примера рассмотрим рис. 25.20.

Операцию запроса getJobs() можно представить следующим образом:

```
context Person::getJobs() : Set(Job)
  body:
    self.Job
```

Выражение self.Job возвращает Set всех объектов Job, ассоциированных с данным объектом Person. Этот Set можно использовать в OCL-выражениях. Пусть согласно бизнес-правилу Person не может занимать две одинаковые должности (Job). На OCL это можно описать так:

```
context Person
  inv:
    -- человек не может занимать две одинаковые должности
    self.Job->isUnique( j : Job | j.name )
```

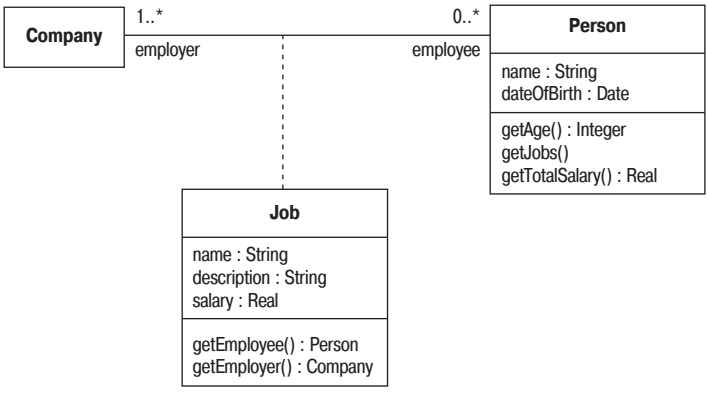


Рис. 25.20. *Навигация к классу-ассоциации по имени*

Предположим, компания (Company) работает по схеме постепенного сокращения круга служебных обязанностей с приближением пенсионного возраста, и существует бизнес-правило, запрещающее человеку (Person) старше 60 занимать более одной должности (Job). На OCL это можно представить следующим образом:

```
context Person
  inv:
    -- человек старше 60 может занимать только одну должность
    (self.getAge() >60) implies (self.Job->count() = 1)
```

Чтобы получить общую заработную плату Person, необходимо сложить его зарплату на каждой должности (Job):

```
context Person::getTotalSalary() : Real
  body:
    -- возвращаем суммарную заработную плату на всех должностях
    self.Job.salary->sum()
```

Навигацию *от* класса-ассоциации можно осуществлять как обычно – с помощью имен ролей на отношениях. Например, вот OCL для операций getEmployee() и getEmployer() класса Job:

```
context Job::getEmployee() : Person
  body:
    self.employee

context Job::getEmployer() : Company
  body:
    self.employer
```

25.12.2. Навигация по квалифицированным ассоциациям

> Для навигации по квалифицированной ассоциации после имени роли в квадратных скобках указываются квалификаторы.

Квалифицированные ассоциации обсуждались в разделе 9.4.6. Для навигации по квалифицированной ассоциации после имени роли в квадратных скобках размещается квалификатор (или разделенный запятыми список квалификаторов).

На рис. 25.21 приведена простая модель клуба с различными уровнями членства. Предположим, существует бизнес-правило, согласно которому членский ID 00001 всегда зарезервирован за председателем клуба. На языке OCL это можно выразить следующим образом:

```
context Club
 inv:
  -- id председателя всегда 00001
    self.members['00001'].level = 'Chairman'
```

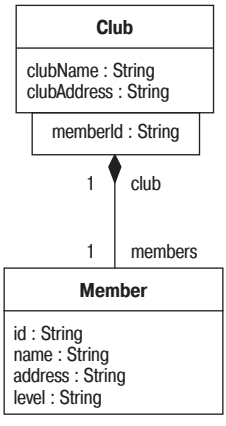


Рис. 25.21. *Модель клуба с различными уровнями членства*

Здесь видно, как используется квалифицированная ассоциация с определенным значением квалификатора для выбора одного объекта из множества (Set) self.members.

25.12.3. Унаследованные ассоциации

Рассмотрим модель на рис. 25.22. Она адаптирована из книги [Arlow 1] и представляет модель единиц измерений и систем единиц.

Есть два разных типа единиц измерения (*Unit*): метрические (*MetricUnits*) и британские (*ImperialUnits*). *MetricUnits* принадлежат метрической системе (MetricSystem), а *ImperialUnits* – британской системе (ImperialSystem). Однако UML-модель на рис. 25.22 *не* отражает этого различия. По сути, в ней определено, что любой *Unit* может принадлежать любой системе единиц (*SystemOfUnits*). Можно сделать модель более точной, если создать подкласс отношения между *SystemOfUnits* и *Unit*, как показано на рис. 25.23.

Это решает проблему, но несколько загромождает диаграмму, и при увеличении числа подклассов *Unit* и *SystemOfUnits* ситуация быстро усугубляется.

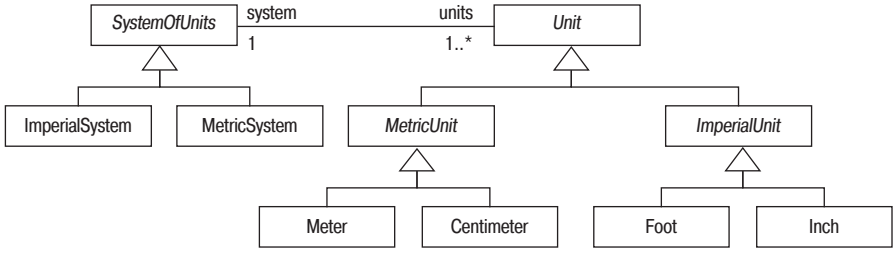


Рис. 25.22. *Модель единиц измерений и систем единиц*

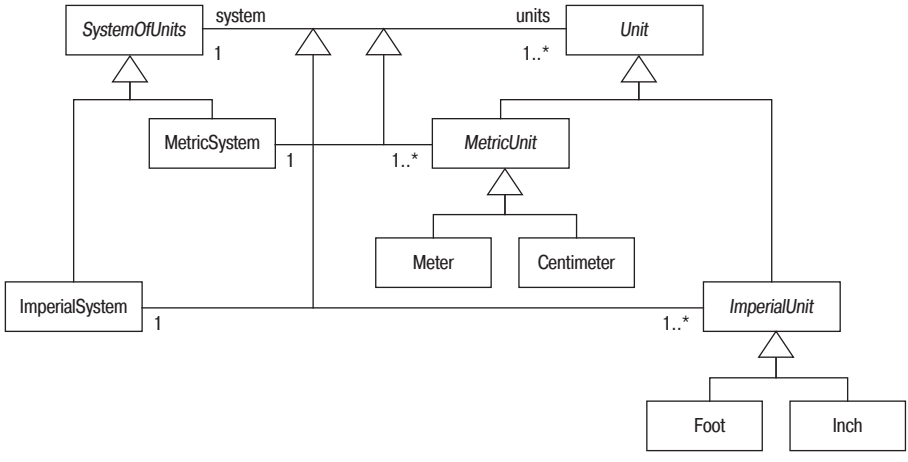


Рис. 25.23. *В модель добавлен подкласс отношения между SystemOfUnits и Unit*

Более изящный способ решения проблемы – определение OCL-ограничений следующим образом:

```
context MetricUnit
  inv:
    -- относится к метрической системе
    self.system.oclIsTypeOf( MetricSystem )

context ImperialUnit
  inv:
    -- относится к британской системе
    self.system.oclIsTypeOf( ImperialSystem )

context MetricSystem
  inv:
    -- все единицы должны быть разновидностью MetricUnit
    -- или одним из его подклассов
    self.units–>forAll( unit | unit.oclIsKindOf( MetricUnit ) )

context ImperialSystem
  inv:
    -- все единицы должны быть разновидностью ImperialUnit
    -- или одним из его подклассов
```

Обратите внимание на использование OCL-выражений oclIsKindOf(...) и oclIsTypeOf(...):

- oclIsKindOf(...) возвращает true, если тип объекта аналогичен типу, определенному параметром, *или* является одним из его подклассов.
- oclIsTypeOf(...) возвращает true, если тип объекта *точно* такой же, как тип, заданный параметром.

25.12.4. OclMessage

> OCL-сообщения могут использоваться только в постусловиях.

OCL-сообщения могут использоваться только в постусловиях. Они позволяют делать следующее:

- подтвердить отправку сообщения;
- подтвердить возвращение сообщения;
- получить возвращаемое значение сообщения;
- вернуть коллекцию сообщений, отправленных объекту.

В OCL каждый вызов операции или отправка сигнала – это экземпляр OclMessage. Он имеет набор операций, приведенный в табл. 25.17.

Таблица 25.17

Операция OclMessage	Семантика
aMessage.isOperationCall() : Boolean	Возвращает true, если сообщение представляет вызов операции.
aMessage.isSignalSent() : Boolean	Возвращает true, если сообщение представляет отправку сигнала.
aMessage.hasReturned() : Boolean	Возвращает true, если сообщение было вызовом операции, которая вернула значение.
aMessage.result() : T	Возвращает результат вызванной операции; T представляет тип возвращаемого значения

Есть также две операции, применяемые к сообщениям. Они приведены в табл. 25.18.

Таблица 25.18

Оператор сообщения	Имя	Семантика
anObject^aMessage()	has sent	Возвращает true, если aMessage() был отправлен в anObject Может использоваться только в постусловиях
anObject^^aMessage()	get messages	Возвращает Sequence сообщений aMessage(), посланных в anObject

Если сообщение имеет параметры, с операторами has sent (послан) и get messages (получить сообщения) может использоваться специальный синтаксис. Это позволяет задавать фактические параметры или только типы параметров. Синтаксис описан в табл. 25.19.

Для изучения сообщений в OCL мы используем шаблон Observer (наблюдатель), полностью описанный в книге [Gamma 1].

Таблица 25.19

has sent с параметрами	**Семантика**
anObject^aMessage(1, 2)	Возвращает true, если aMessage(...) послано в anObject с параметрами, имеющими значения 1 и 2.
anObject^aMessage(?:Integer, ?:Integer)	Возвращает true, если aMessage(...) послано в anObject с параметрами типа Integer.
get messages с параметрами	**Семантика**
anObject^^aMessage(1, 2)	Возвращает Sequence всех сообщений, посланных в anObject с параметрами, имеющими значения 1 и 2.
anObject^^aMessage(?:Integer, ?:Integer)	Возвращает Sequence всех сообщений, посланных в anObject с параметрами типа Integer.

Простой пример применения шаблона Observer показан на рис. 25.24. Семантика этого шаблона проста: субъект (Subject) имеет нуль или более наблюдателей (Observer). При изменении Subject вызывается его операция notify() (уведомить), которая в свою очередь вызывает операцию update() (обновить) каждого прикрепленного к ней Observer. Примером обычного использования этого шаблона может быть обновление экрана при изменении базовых бизнес-объектов.

Рис. 25.24. *Пример применения шаблона Observer*

Семантика операций Subject представлена в табл. 25.20.

Таблица 25.20

Операция Subject	**Семантика**
attach(o : Observer)	Присоединяет объект Observer.
detach(o : Observer)	Отсоединяет объект Observer.
notify()	Вызывает операцию update() каждого объекта Observer – вызывается, когда Subject меняется и желает уведомить объекты Observer об этом изменении.

Рассмотрим возможные постусловия для операций Subject, которые могут использовать сообщения.

Согласно табл. 25.20 постусловие Subject::notify() состоит в том, что Observer::update() был вызван для каждого присоединенного Observer. На языке OCL это может быть выражено следующим образом:

```
context Subject::notify( )
```

```
post:
 -- каждый наблюдатель должен быть уведомлен через вызов update()
 self.observers.forAll( observer | observer^update() )
```

В данном постусловии происходит перебор всех observers в Set и проверка получения ими сообщения update(). Оператор ^ – это оператор «has sent». Он возвращает true, если указанное сообщение было послано операцией. Он может использоваться только в постусловиях.

Если сообщение синхронное (вы знаете, что оно вернется), можно проверять возвращаемое значение сообщения:

```
context Subject::notify( )
 post:
  -- возвращаемое значение update( ) должно быть true для каждого наблюдателя
  self.observers.forAll( observer |
                         observer^update().hasReturned() implies
                         observer^update().result()
                         )
```

25.13. Что мы узнали

В этом введении в OCL мы узнали следующее:

- OCL – стандартное расширение UML. Он может:
  - определять запросы;
  - определять ограничения;
  - определять тело операции запроса;
  - определять бизнес-правила во время разработки модели.
- OCL *не* язык действий для UML. Он не может:
  - менять значения элемента модели;
  - определять тело операции (кроме операций запроса);
  - выполнять операции (кроме операций запроса);
  - динамически определять бизнес-правила.
- OCL-выражения.
  - Состоят из:
    - контекста пакета (необязательно) – определяет пространство имен для OCL-выражения;
    - контекста выражения (обязательно) – определяет экземпляр контекста для выражения;
    - одного или более выражений.
  - Контекст выражения может быть определен явно или путем размещения OCL-выражения в примечании к элементу модели.

- Экземпляр контекста – образец экземпляра контекста выражения:
  - OCL-выражения записываются в рамках экземпляра контекста.
- Типы OCL-выражений.
  - inv: – инвариант:
    - применяется к классификаторам;
    - экземпляр контекста – экземпляр классификатора;
    - инвариант должен быть true для всех экземпляров классификатора.
  - pre: – предусловие операции:
    - применяется к операциям;
    - экземпляр контекста – экземпляр классификатора, которому принадлежит операция;
    - предусловие должно выполняться для обеспечения возможности выполнения операции.
  - post: – постусловие операции:
    - применяется к операциям;
    - экземпляр контекста – экземпляр классификатора, которому принадлежит операция;
    - постусловие должно быть истинным после выполнения операции;
    - ключевое слово result обозначает результат операции;
    - feature@pre возвращает значение свойства (feature) до выполнения операции:
      - @pre может использоваться только в постусловиях;
    - OclMessage может использоваться только в постусловиях.
  - body: – определяет тело операции:
    - применяется к операциям запроса;
    - экземпляр контекста – экземпляр классификатора, которому принадлежит операция.
  - init: – задает начальное значение:
    - применяется к атрибутам или концам ассоциаций;
    - экземпляр контекста – атрибут или конец ассоциации.
  - def: – добавляет атрибуты и операции в классификатор с помощью стереотипа «OclHelper»:
    - применяется к классификатору;
    - экземпляр контекста – экземпляр классификатора;
    - добавляет атрибуты и операции в классификатор с помощью стереотипа «OclHelper»:
      - может использоваться только в OCL-выражениях.

    - let – определяет локальную переменную:
        - применяется к OCL-выражению;
        - экземпляр контекста OCL-выражения.
    - derive: – определяет производное значение:
        - применяется к атрибутам или концам ассоциаций;
        - экземпляр контекста – экземпляр классификатора, которому принадлежит операция.
- Комментарии:
    - /* многострочные */
    - -- однострочные
- Старшинство операций:
    - ::
    - @pre
    - .
    - ->
    - not - ^ ^^
    - * /
    - + -
    - if then else endif
    - > < <= >=
    - = <>
    - and or xor
    - implies
- Система типов OCL.
    - OclAny:
        - супертип всех типов OCL и типов UML-модели.
    - OclType:
        - подкласс OclAny;
        - перечисление всех типов ассоциированной UML-модели.
    - OclState:
        - подкласс OclAny;
        - перечисление всех состояний ассоциированной UML-модели.
    - OclVoid:
        - тип «null» в OCL – имеет единственный экземпляр под названием OclUndefined.
    - OclMessage – используется для:
        - подтверждения отправки сообщения;

    - подтверждения возвращения сообщения;
    - получения возвращаемого значения сообщения;
    - возвращения коллекции сообщений, отправленных объекту.
  - Примитивные типы OCL:
    - Boolean;
    - Integer;
    - Real;
    - String.
  - Tuples – структурированный тип:
    - Tuple { имяЧасти1:типЧасти1 = значение1, имяЧасти2:типЧасти2 = значение2, … }
- Инфиксные операторы – для их добавления в типы UML просто определяется операция с соответствующей сигнатурой, например =(параметр) : Boolean.
- Коллекции:
  - Set – { unordered, unique } (применяется по умолчанию)
  - OrderedSet – { ordered, unique }
  - Bag – { unordered, nonunique }
  - Sequence – { ordered, nonunique }
- Операции над коллекциями должны инициироваться с помощью оператора –>.
- Итерационные операции:
  - коллекция–> <операцияИтератора>(<переменнаяИтератора>:<Тип> | <выражениеИтератора>);
  - <переменнаяИтератора> и <Тип> необязательны.
- Навигация в рамках экземпляра контекста:
  - self – организует доступ к экземпляру контекста;
  - оператор «точка» – организует доступ к возможностям экземпляра контекста.
- Навигация по ассоциациям:
  - self – организует доступ к экземпляру контекста;
  - имя роли – обозначает объект или набор объектов на конце ассоциации;
  - оператор «точка» – перегружен:
    - self.roleName:
      - кратность 1 – обеспечивается доступ к объекту на конце ассоциации;
      - кратность >1 – обеспечивается доступ к коллекции на конце ассоциации;

    - self.roleName.feature:
        - кратность 1 – возвращает значение свойства (feature);
        - кратность > 1 – возвращает Bag значений свойств всех участвующих объектов (является сокращенной записью для self.roleName->collect(feature)).
- Навигация по нескольким ассоциациям, где *все* кратности > 1:
    - self.roleName1.roleName2.feature – возвращает Bag значений этого свойства (feature) всех участвующих объектов.
- OCL на диаграммах взаимодействий используется для определения:
    - сторожевого условия;
    - селектора линии жизни;
    - параметров сообщения.
- OCL на диаграммах деятельностей используется для определения:
    - узлов вызова действий;
    - сторожевых условий переходов;
    - объектных узлов;
    - операций;
    - состояния объекта.
- OCL на диаграммах состояний используется для определения:
    - сторожевых условий;
    - условий, накладываемых на состояния;
    - целей действий;
    - операций;
    - значений параметров.
- Навигация к и от классов-ассоциаций:
    - к классам-ассоциаций – используется имя класса-ассоциации;
    - от классов-ассоциаций – используются имена ролей.
- Навигация по квалифицированным ассоциациям:
    - после имени роли в квадратных скобках указывается разделенный запятыми список квалификаторов.
- Унаследованные ассоциации:
    - OCL-выражения используются для ограничения типов участников в унаследованных ассоциациях.

A

Пример модели прецедентов

A.1. Введение

Наш опыт свидетельствует о том, что UML-модели не терпят бумаги. Если вам когда-либо приходилось распечатывать большую UML-модель, включая спецификации, вам точно известно, что имеется в виду! UML-модели лучше представлять в более гибком формате – в виде гипертекста. В настоящее время это или средство моделирования, или веб-сайт.

Включение полного учебного примера UML в эту книгу сделало бы ее намного толще и дороже. Мы также несли бы ответственность за намного большее количество уничтоженных деревьев. Поэтому было решено предоставить пример для этой книги на нашем веб-сайте (*www.umlandtheunifiedprocess.com*). Нам кажется, что ориентироваться в электронном варианте легче, чем в бумажном.

Предлагаемый вашему вниманию пример охватывает ОО анализ и проектирование, необходимые для создания небольшого веб-приложения электронной коммерции. В этом приложении в упрощенном виде приводится несколько наиболее ярких моментов модели прецедентов, чтобы дать представление о том, что доступно на сайте!

A.2. Модель прецедентов

Данная модель прецедентов создана для простой системы электронной коммерции по продаже книг и CD. Эта система называется ECP (E-Commerce Platform – платформа электронной коммерции). На рис. A.1 показан окончательный результат моделирования прецедентов.

Модель прецедентов обеспечит четкое представление о том, что делает система, но все подробности представлены на веб-сайте.

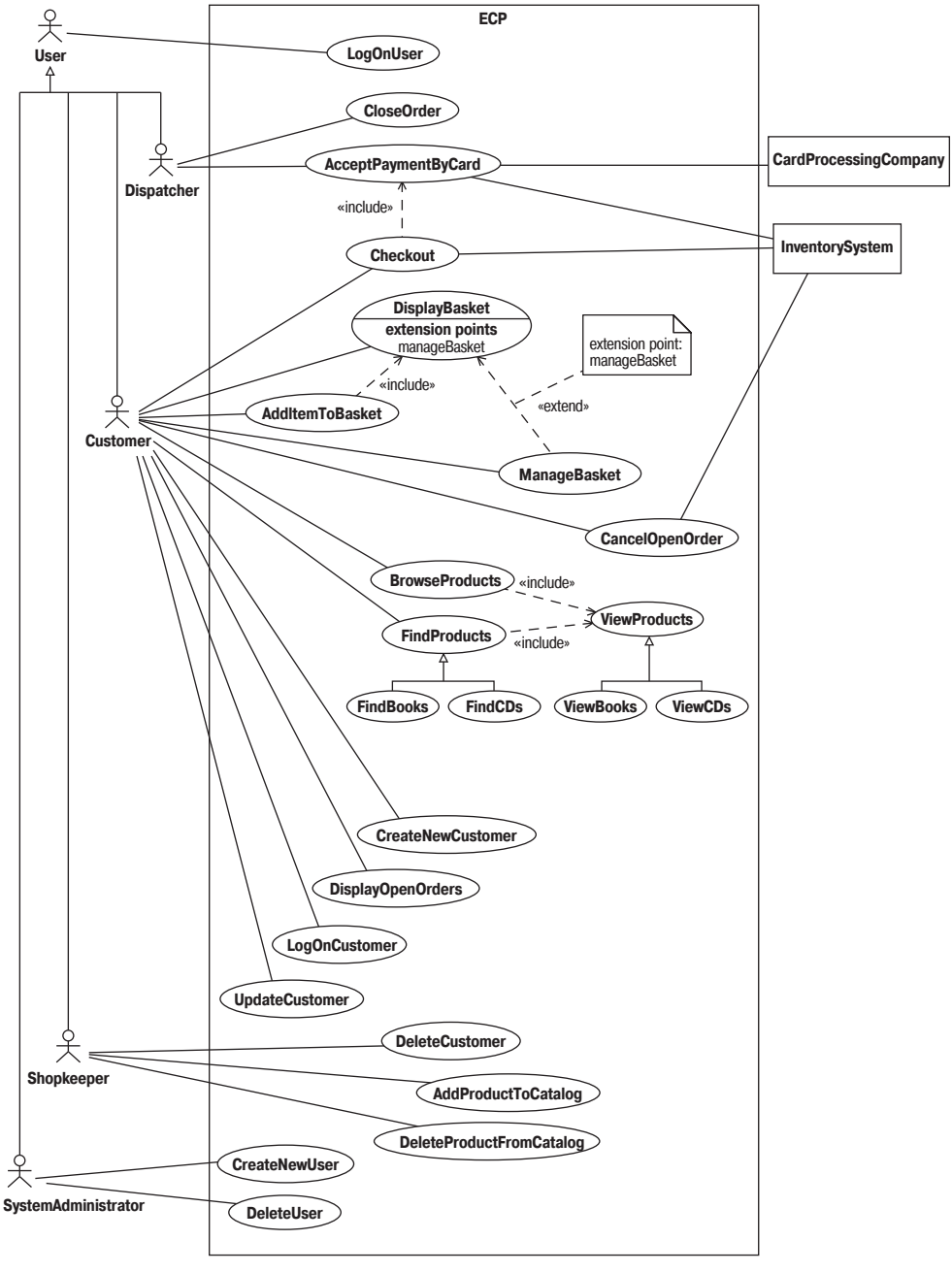


Рис. А.1. *Модель прецедентов*

А.3. Примеры прецедентов

На рис. А.2 представлен сокращенный вариант модели прецедентов. Здесь показаны обычные прецеденты, расширяющий прецедент и отношения «include» и «extend».

Прецеденты с рис. А.2 показаны более подробно на рис. А.3–А.6.

В описания прецедентов включены все важные детали, но опущена общая информация (торговая марка компании, информация об авторе и версии и др.). Эти данные для каждой компании свои. Во многих компаниях разработаны стандартные заголовки, используемые во всей документации компании.

Хотя описание прецедента может храниться непосредственно в инструментальном средстве моделирования UML, часто поддержка этой возможности довольно слаба и ограничивается простым текстом. Поэтому многие разработчики моделей сохраняют описания прецедентов в формате, предоставляющем бо́льшие возможности, таком как Word или XML, и подключаются к этим внешним документам из модели прецедентов в инструменте моделирования. Некоторые идеи по поводу

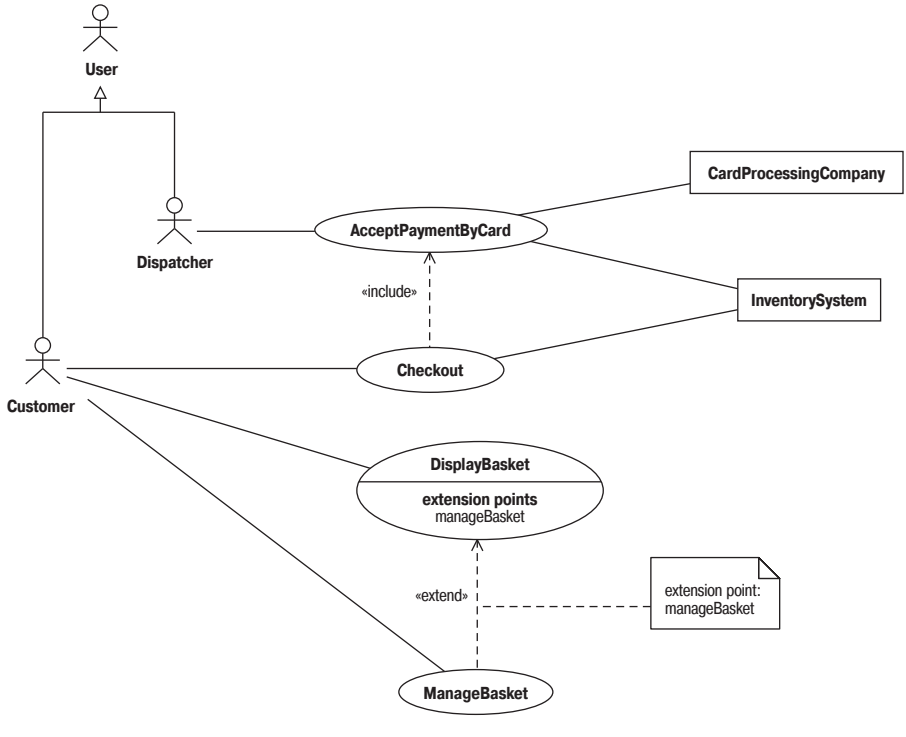


Рис. А.2. *Сокращенный вариант модели прецедентов*

использования XML для записи описаний прецедентов представлены в приложении B.

Прецедент: Checkout
ID: 6
Краткое описание: Customer (покупатель) подтверждает заказ. Система создает заказ на основании содержимого корзины для покупок, и Customer оплачивает заказ.
Главные актеры: Customer
Второстепенные актеры: InventorySystem (система управления запасами)
Предусловия: 1. Customer входит в систему.
Основной поток: 1. Прецедент начинается, когда Customer выбирает опцию «checkout». 2. Система просит актера InventorySystem предварительно зарезервировать товарные позиции, указанные в корзине для покупок. 3. Для каждой отсутствующей позиции. 3.1. Система информирует Customer о том, что товар временно недоступен и удален из заказа. 4. Система представляет окончательный вариант заказа актеру Customer. Для каждого продукта заказ включает идентификатор продукта, имя продукта, количество, цену единицы продукции, общую стоимость данного количества. В заказ также входит адрес поставки, информация кредитной карты Customer и общая стоимость заказа, включая налоги и затраты на доставку и упаковку. 5. Система просит Customer принять или отклонить заказ. 6. Customer подтверждает заказ. 7. include(AcceptPaymentByCard).
Постусловия: 1. Customer подтвердил заказ. 2. Заказанные товары зарезервированы актером InventorySystem.
Альтернативные потоки: Нет.

Рис. A.3. *Описание прецедента Checkout*

Прецедент: AcceptPaymentByCard
ID: 1
Краткое описание: Покупатель оплачивает заказ кредитной картой.
Главные актеры: Customer
Второстепенные актеры: CardProcessingCompany (компания обработки кредитных карт) InventorySystem (система управления запасами) Dispatcher (диспетчер)
Предусловия: 1. Customer входит в систему. 2. Некоторые наличные товарные позиции были предварительно зарезервированы для актера Customer.
Основной поток: 1. Прецедент начинается, когда Customer подтверждает заказ. 2. Система извлекает информацию кредитной карты Customer. 3. Система посылает сообщение в CardProcessingCompany, включающее идентификатор получателя платежа, его аутентификационные данные, имя на карте, номер карты, срок действия карты, сумму сделки. 4. CardProcessingCompany дает разрешение на транзакцию. 5. Система сообщает Customer, что транзакция с использованием кредитной карты была принята. 6. Система дает Customer шифр, чтобы он мог отслеживать заказ. 7. Система указывает актеру InventorySystem зарезервировать необходимые товарные позиции. 8. Система посылает заказ актеру Dispatcher. 9. Система меняет состояние заказа на ожидающий рассмотрения. 10. Система выводит на экран подтверждение заказа, предоставляя актеру Customer возможность распечатать его.
Постусловия: 1. Заказ получил статус ожидающего рассмотрения. 2. С кредитной карты Customer снята соответствующая сумма. 3. Некоторые наличные товарные позиции были зарезервированы для обеспечения выполнения заказа. 4. Заказ отправлен актеру Dispatcher.
Альтернативные потоки: CreditLimitExceeded BadCard CreditCardPaymentSystemDown

Рис. А.4. *Описание прецедента AcceptPaymentByCard*

Прецедент: DisplayBasket
ID: 13
Краткое описание: Система отображает содержимое корзины для покупок актера Customer.
Главные актеры: Customer
Второстепенные актеры: Нет.
Предусловия: Нет.
Основной поток: 1. Customer выбирает опцию «display basket» (вывести на экран содержимое корзины). 2. Если в корзине нет товаров. 2.1. Система сообщает Customer о том, что корзина пуста. 2.2. Прецедент завершается. 3. Для каждого продукта в корзине. 3.1. Система отображает идентификационный номер, количество, детальную информацию, цену единицы продукции и общую цену. точка расширения: manageBasket
Постусловия: Нет.
Альтернативные потоки: Нет.

Рис. А.5. *Описание прецедента DisplayBasket*

Расширяющий прецедент: ManageBasket
ID: 20
Краткое описание: Customer меняет содержимое корзины.
Главные актеры: Customer
Второстепенные актеры: Нет.
Предусловия: 1. Система отображает корзину для покупок.
Основной поток: 1. Пока Customer вносит изменения в корзину. 1.1. Customer выбирает товарную позицию в корзине. 1.2. Если Customer выбирает «remove item» (удалить позицию). 1.2.1. Система отображает сообщение «Вы уверены, что хотите удалить из корзины выбранную позицию?». 1.2.2. Customer подтверждает удаление. 1.2.3. Система удаляет выбранную позицию из корзины. 1.3. Если Customer вводит новое количество для выбранной позиции. 1.3.1. Система обновляет количество для выбранной позиции.
Постусловия: Нет.
Альтернативные потоки: Нет.

Рис. А.6. *Описание прецедента ManageBasket*

B

XML и прецеденты

B.1. Применение XML для шаблонов прецедентов

Как видите, UML 2 не определяет формального стандарта для документирования прецедентов. Разработчикам моделей приходится или использовать возможности, предлагаемые инструментальными средствами моделирования UML, которые нередко весьма ограничены, или применять собственный подход. В настоящий момент модель прецедентов чаще всего создается в средстве моделирования, а затем прецеденты и актеры подключаются к внешним документам, содержащим их подробные описания. Эти документы обычно создаются в текстовом редакторе, однако это не лучший инструмент для решения данной задачи. Хотя он обеспечивает возможность форматирования и структурирования описаний прецедентов и актеров, в нем нельзя отразить семантику этой структуры.

Мы считаем, что структурированные XML-документы – естественный формат описаний прецедентов. XML – это язык семантической разметки, поэтому в нем семантическая структура документа отделена от его форматирования. Описание прецедента или актера, представленное в виде XML-документа, можно трансформировать разными способами с помощью XSL [Kay 1]. Описания можно генерировать как HTML, PDF или как документы текстового редактора. Кроме того, можно делать запросы к описаниям для получения определенной информации.

Структура XML-документа может быть описана на языке описания XML-схемы (XML Schema Definition Language, XSL). Несколько простых XML-схем для актеров и прецедентов представлены на нашем вебсайте. Они доступны для загрузки и использования по лицензии GNU (GNU General Public License) (подробности см. по адресу *www. gnu.org*).

Подробное описание XML и XSL выходит за рамки как этой книги, так и нашего веб-сайта. Однако на сайте предлагаются полезные ссылки на образовательные ресурсы по XML и XSL.

Поскольку применение XML требует специальных редакторов, и заинтересованным сторонам может быть сложно его использовать, недавно нами был разработан простой подход создания прецедентов (и других документов проекта) в XML и других форматах. Он кратко обсуждается в следующем разделе.

B.2. SUMR

SUMR (произносится «саммэ») расшифровывается как Simple Use case Markup-Restructured (простая реструктурированная разметка прецедентов). Это текстовый язык разметки для прецедентов. Документы SUMR без труда можно трансформировать в XML, HTML и другие форматы.

SUMR обладает целым рядом преимуществ.

- Он *очень* прост – синтаксис разметки можно выучить за минуту.
- Для него не требуются изощренные текстовые редакторы или возможности редактирования – SUMR-документы можно создавать в любом текстовом редакторе, поддерживающем простой текст. В случае необходимости SUMR-текст можно вводить прямо в поле HTML-формы.
- Он структурированный – SUMR-документы имеют схему. Можно создать собственную схему или использовать предоставляемые нами стандартные схемы.
- Он бесплатен согласно разрешению для копирования GNU (*www.gnu.org/copyleft*).

Чтобы вы почувствовали, что такое SUMR, на рис. B.1 приведен простой прецедент, размеченный как SUMR-документ.

Синтаксис прост: все, что начинается и заканчивается двоеточиями (например, :это:), – это тег. Тело тега начинается на следующей строке после тега и продолжается до следующей пустой строки.

SUMR занимается исключительно структурой и семантикой документа, а *не* его представлением. Он позволяет быстро и эффективно фиксировать важную информацию, не забивая голову форматированием, шаблонами документов или сложными языками разметки.

У каждого SUMR-документа может быть схема, определенная в специальном теге :schema:. Схема SUMR – SUMR-документ, определяющий теги, которые могут использоваться в других SUMR-документах. На рис. B.2 показана SUMR-схема нашего примера прецедента. Как видите, правильно описанные схемы SUMR могут быть самодокументируемыми.

```
file AddItemToBasket.uc

:schema:
UseCase.sss

:name:
AddItemToBasket

:id:
2

:parents:
1. None

:primaryActors:
1. Customer

:secondaryActors:
1. None

:brief:
1. The Customer adds an item to their shopping basket.

:pre:
1. The Customer is browsing products.

:flow:
1. The Customer selects a product
2. The Customer selects "add item".
3. The system adds the item to the Customer's shopping basket.
4. :inc:DisplayBasket

:post:
1. A product has been added to the Customer's basket.
2. The contents of the basket are displayed.

:alt:
1. None

:req:
1. None
```

Рис. B.1. *Прецедент, размеченный как SUMR-документ*

Когда есть схема, инструменты SUMR можно использовать для:

- трансформирования прецедентов, соответствующих схеме, в XML;
- генерирования образцовых таблиц стилей XSL для преобразования XML в
  - XHTML плюс каскадная таблица стилей (Cascading Style Sheet, CSS);
  - XML-FO (форматирующие объекты XML).

```
file UseCase.sss

:schema:
UseCase.sss

:name:
Write the name of the use case here. Use case names are
in UpperCamelCase with no spaces.

:id:
Write the unique project identifier for the use case here.

:parents:
Write the names of the parent use cases here.
If this use case has no parents, write None here.

:primaryActors:
Write the names of the primary actors here.
There must be at least one primary actor.

:secondaryActors:
List the names of the secondary actors here.
Secondary actors participate in the use case, they do not start the use case.
If there are no secondary actors, write None here.

:brief:
Write a brief description of your use case here. This description
should be no more than a couple of paragraphs.

:pre:
Write the preconditions here, one on each line.
If the use case has no preconditions, write None here.

:flow:
Write the main flow here.
Each step should be time-ordered and declarative.
:ext:WriteExtensionPointsLikeThis
Note that extension points are NOT numbered.
If you need to show nested steps
  Indent them by one space for each level of indent like this.
Include other use cases like this :inc:AnotherUseCase. This is the final step.

:post:
Write the postconditions here, one on each line.
If there are no postconditions, write None here.

:alt:
List the names of the alternative flows here, one on each line.
If there are no alternative flows, write None here.

:req:
List any special requirements related to the use case here. These are
typically non-functional requirements.
If there are no special requirements, write None here.
```

***Рис. В.2.** SUMR-схема прецедента*

Инструменты SUMR генерируют стандартные таблицы стилей. Это полезная возможность – не нужно писать таблицы стилей с нуля, достаточно только настроить их. Это также очень гибкий подход, поскольку таблицы стилей предоставляют практически полный контроль над формированием представления SUMR-документов.

На рис. B.3 представлен пример прецедента с рис. B.1, трансформированного в XML одним из инструментов SUMR.

Как только прецедент представлен в формате XML, можно использовать сгенерированные таблицы стилей XSL для его дальнейшей трансформации. Вариант, обеспечивающий максимальную гибкость, – это трансформация XML в XML-FO; после этого с помощью Apache FOP (*http://xml.apache.org/fop/index.html*) его можно преобразовать во множество других форматов вывода, включая PDF и PostScript. На рис. B.4 показан прецедент, сформированный как PDF с помощью стандартной таблицы стилей XML-FO, сгенерированной из схемы. В стандартном стиле практически не на что смотреть, но его можно как угодно настроить, редактируя сгенерированную таблицу стилей, которая его описывает.

И наконец, на рис. B.5 представлен простой структурированный редактор документов SUMR, который мы включили в набор инструментов. Он может проверять соответствие прецедента или актера его схеме, имеет средства подсветки синтаксиса и автонумерации. Это замечательный инструмент для малых и средних моделей прецедентов.

Найти более подробную информацию по SUMR и загрузить инструменты и примеры схем можно на нашем веб-сайте (*www.umlandtheunifiedprocess.com*).

Надеемся, вам понравится работать с этим набором инструментальных средств. Для нас ценен любой отзыв, который вы можете оставить на нашем веб-сайте.

file AddItemToBasket.uc

```
<?xml version="1.0" encoding="UTF-8"?>
<UseCase.sss>
  <schemaSection>
    <schema>UseCase.sss</schema>
  </schemaSection>
  <nameSection>
    <name>AddItemToBasket</name>
  </nameSection>
  <idSection>
    <id>2</id>
  </idSection>
  <parentsSection>
    <parents>1. None</parents>
  </parentsSection>
  <primaryActorsSection>
    <primaryActors>1. Customer</primaryActors>
  </primaryActorsSection>
  <secondaryActorsSection>
    <secondaryActors>1. None</secondaryActors>
  </secondaryActorsSection>
  <briefSection>
    <brief>1. The Customer adds an item to their shopping
              basket.</brief>
  </briefSection>
  <preSection>
    <pre>1. The Customer is browsing products.</pre>
  </preSection>
  <flowSection>
    <flow>1. The Customer selects a product</flow>
    <flow>2. The Customer selects "add item".</flow>
    <flow>3. The system adds the item to the Customer's shopping
             basket.</flow>
    <flow>4. :inc:DisplayBasket</flow>
  </flowSection>
  <postSection>
    <post>1. A product has been added to the Customer's basket.</post>
    <post>2. The contents of the basket are displayed.</post>
  </postSection>
  <altSection>
    <alt>1. None</alt>
  </altSection>
  <reqSection>
    <req>1. None</req>
  </reqSection>
</UseCase.sss>
```

Рис. В.3. *Прецедент с рис. В.1, трансформированный в XML одним из инструментов SUMR*

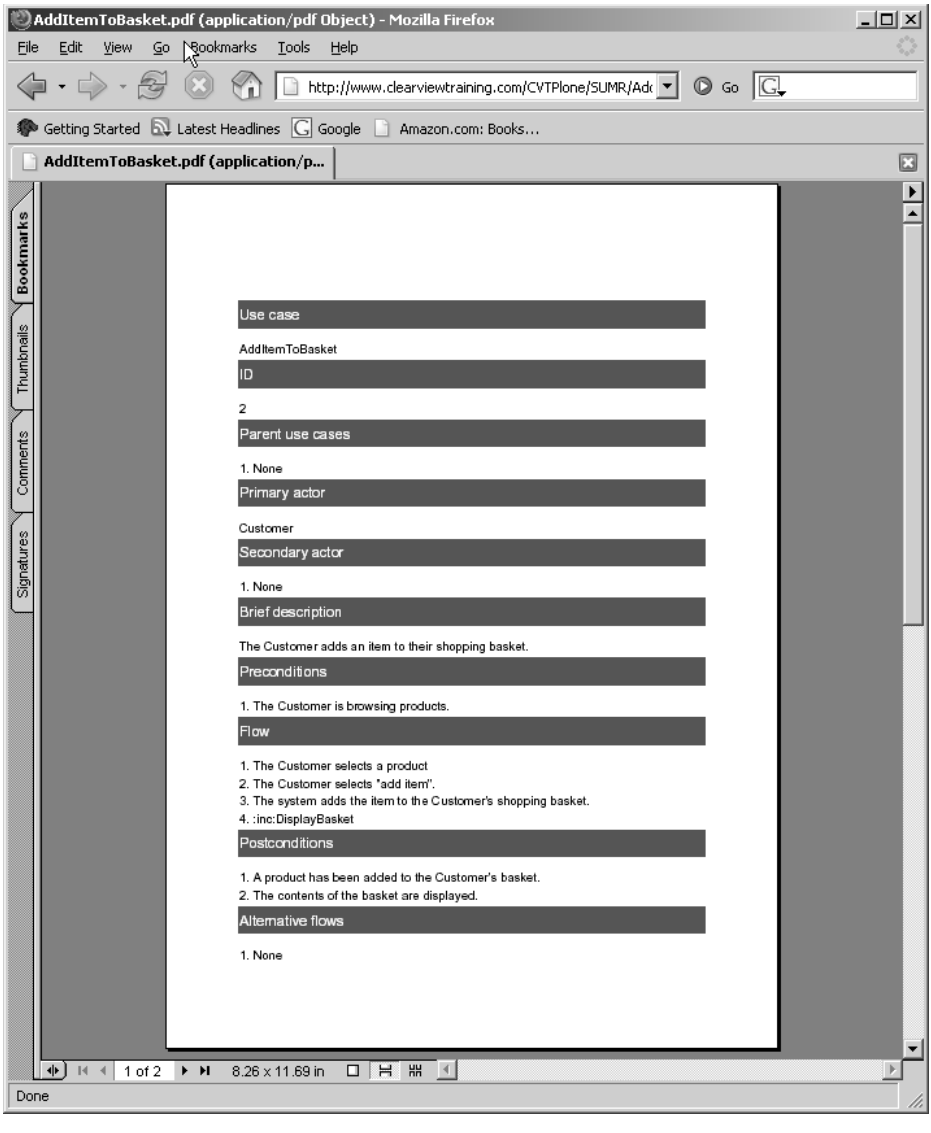


Рис. B.4. *Прецедент в формате PDF*

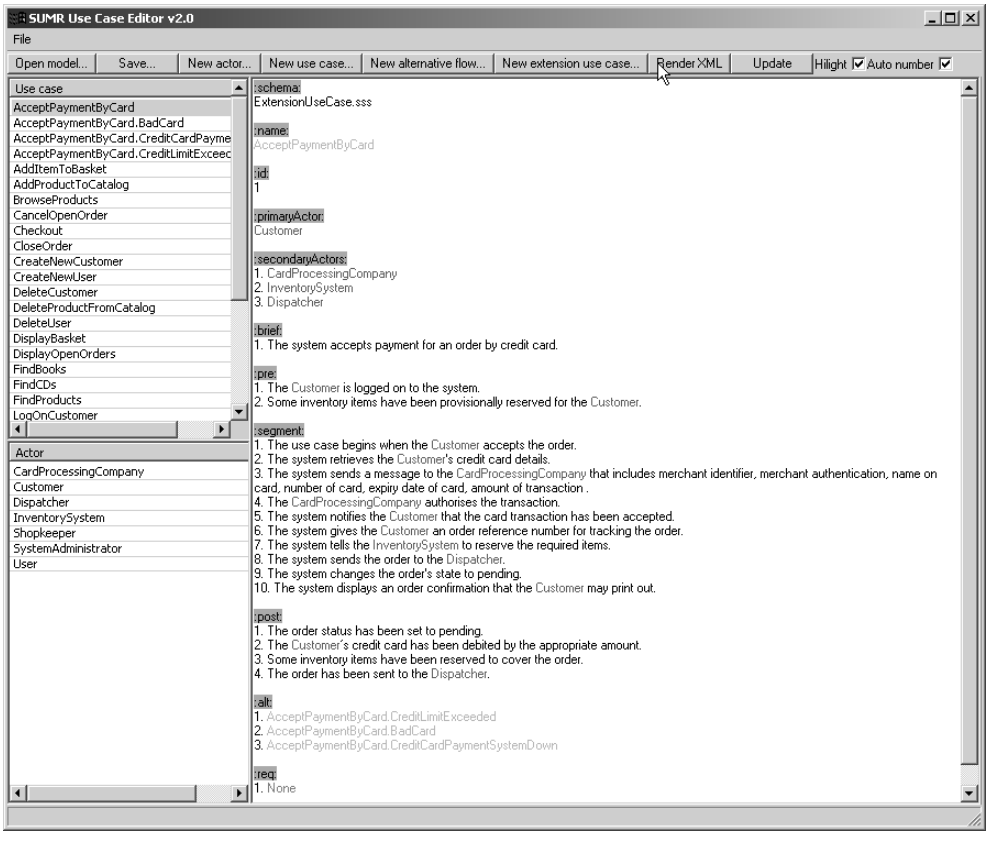


Рис. B.5. *Структурированный редактор документов SUMR*

Библиография

[Alexander 1], «Writing Better Requirements», Ian F. Alexander, Richard Stevens, Ian Alexander, Addison-Wesley, 2002, ISBN 0321131630.

[Ambler 1], «The Unified Process Inception Phase», Scott W. Ambler, Larry L. Constantine, CMP Books, 2000, ISBN 1929629109.

[Ambler 2], «The Unified Process Elaboration Phase», Scott W. Ambler, Larry L. Constantine, Roger Smith, CMP Books, 2000, ISBN 1929629052.

[Ambler 3], «The Unified Process Construction Phase», Scott W. Ambler, Larry L. Constantine, CMP Books, 2000, ISBN 192962901X.

[Arlow 1], «Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML», Jim Arlow, Ila Neustadt, Addison-Wesley, 2004, ISBN 032111230X.

[Booch 1], «Object Solutions», Grady Booch, Addison-Wesley, 1995, ISBN 0805305947.

[Booch 2], «The Unified Modeling Language User Guide», Grady Booch, Ivar Jacobson, James Rumbaugh, Addison-Wesley, 1998, ISBN 0201571684.

[Chomsky 1], «Syntactic Structures», Noam Chomsky, Peter Lang Publishing, 1975, ISBN 3110154129.

[Frankel 1], «Model Driven Architecture: Applying MDA to Enterprise Computing», David S. Frankel, John Wiley & Sons, 2003, ISBN 0471319201.

[Gamma 1], «Design Patterns», Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley, 1995, ISBN 0201633612.[1]

[Harel 1], «Modeling Reactive Systems with Statecharts: The Statemate Approach», David Harel, Michal Politi, McGraw Hill, 1998, ISBN 0070262055.

[1] Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес «Приемы объектно-ориентированного проектирования. Паттерны проектирования». – Пер. с англ. – СПб.: Питер, 2007.

[Jacobson 1], «Unified Software Development Process», Ivar Jacobson, Grady Booch, James Rumbaugh, Addison-Wesley, 1999, ISBN 0201571692.

[Kay 1], «XSLT Programmer's Reference», 2nd Edition, Michael Kay, Wrox Press Inc., 2001, ISBN 1861005067.[1]

[Kleppe 1], «MDA Explained: The Model Driven Architecture – Practice and Promise», Anneke Kleppe, Jos Warmer, Wim Bast, Addison-Wesley, 2001, ISBN 032119442X.

[Kroll 1], «The Rational Unified Process Made Easy: A Practitioner's Guide to Rational Unified Process», Per Kroll, Philippe Kruchten, Grady Booch, Addison-Wesley, 2003, ISBN 0321166094.

[Kruchten 1], «The Rational Unified Process, An Introduction», Philippe Kruchten, Addison-Wesley, 2000, ISBN 0201707101.

[Kruchten 2], «The 4+1 View of Architecture», Philippe Kruchten, IEEE Software, 12(6) Nov. 1995, pp. 45–50.

[Leffingwell 1], «Managing Software Requirements: A Use Case Approach», 2nd Edition, Dean Leffingwell, Don Widrig, Addison-Wesley, 2003, ISBN 032112247X.

[Meyer 1], «Object Oriented Software Construction», Bertrand Meyer, Prentice Hall, 1997, ISBN 0136291554.

[Mellor 1], «Agile MDA, Stephen J Mellor, MDA Journal», June 2004, *www.bptrends.com*.

[OCL1], «Unified Modeling Language: OCL, version 2.0», *www.omg.org*.

[Pitts 1], «XML Black Book», 2nd Edition, Natalie Pitts, Coriolis Group, 2000, ISBN 1576107833.

[Rumbaugh 1], «The Unified Modeling Language Reference Manual», 2nd Edition, James Rumbaugh, Ivar Jacobson, Grady Booch, Addison-Wesley, 2004, ISBN 0321245628.

[Standish 1], «The CHAOS Report (1994)», The Standish Group, *www.standishgroup.com/sample_research/chaos_1994_1.php*.

[UML2S], «Unified Modeling Language: Superstructure, version 2.0», *www.omg.org*.

[Warmer 1], «The Object Constraint Language: Getting Your Models Ready for MDA», Jos Warmer, Anneke Kleppe, Addison-Wesley, 2003, ISBN 0321179366.

[1] Майкл Кэй «XSLT. Справочник программиста», 2-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2002.

Алфавитный указатель

P

R

S

T

U

V

W

X

A

Б

В

Г

Д

З

И

Л

М

Н

О

П

Р

C

Т

У

Ф

Ц

Ч

Ш

Э

Я