Гради Буч, Джеймс Рамбо, Ивар Якобсон

Язык UML. Руководство пользователя

The Unified Modeling Language Usere Guide

SECOND EDITION

Grady Booch
James Rumbaugh
Ivar Jacobson

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Язык UML Руководство пользователя

ВТОРОЕ ИЗДАНИЕ

Гради Буч
Джеймс Рамбо
Ивар Якобсон

АКАДЕМИЯ АЙТИ ДМК ПРЕСС издательство

Москва, 2006

УДК 004.434:004.94UML
ББК 32.973.26-018.1
Б90

Буч Г., Рамбо Д., Якобсон И.
Б90 Язык UML. Руководство пользователя. 2-е изд.: Пер. с англ. Мухин Н. – М.: ДМК Пресс, 2006. – 496 с.: ил.

ISBN 5-94074-334-X

Унифицированный язык моделирования (Unified Modeling Language, UML) является графическим языком для визуализации, специфицирования, конструирования и документирования систем, в которых большая роль принадлежит программному обеспечению. С помощью UML можно разработать детальный план создаваемой системы, содержащий не только ее концептуальные элементы, такие как системные функции и бизнес-процессы, но и конкретные особенности, например классы, написанные на специальных языках программирования, схемы баз данных и программные компоненты многократного использования.

Предлагаемое вашему вниманию руководство пользователя содержит справочный материал, дающий представление о том, как можно использовать UML для решения разнообразных проблем моделирования. В книге подробно, шаг за шагом, описывается процесс разработки программных систем на базе данного языка.

Издание адресовано читателям, которые уже имеют общее представление об объектно-ориентированных концепциях (опыт работы с конкретными объектно-ориентированными языками или методиками не требуется, хотя желателен). В первую очередь руководство предназначено для разработчиков, занятых созданием моделей UML/ Тем не менее, книга будет полезна всем, кто осваивает, создает, тестирует или выпускает в свет программные системы.

УДК 004.434:004.94UML
ББК 32.973.26-018.1

ISBN 0-321-26797-4 (англ.)
ISBN 5-94074-334-X (рус.)

Содержание

Введение

Унифицированный язык моделирования (Unified Modeling Language, UML) – это графический язык для визуализации, специфицирования, конструирования и документирования систем, в которых главная роль принадлежит программному обеспечению. С помощью UML можно разработать детальный план создаваемой системы, содержащий не только ее концептуальные элементы, такие как системные функции и бизнес-процессы, но и конкретные особенности, например классы, написанные на каком-либо языке программирования, схемы баз данных и повторно используемые программные компоненты.

Эта книга научит вас эффективно использовать UML. В ней рассматривается версия UML 2.0.

Цели

Прочитав эту книгу, вы:

- научитесь различать, в чем может, а в чем не может пригодиться UML, а также поймете, почему этот язык важен в процессе разработки сложных программных систем;
- освоите словарь, правила, идиомы языка UML и, самое главное, научитесь «бегло говорить» на нем;
- поймете, как можно использовать UML для решения разнообразных проблем моделирования.

Предлагаемое вашему вниманию руководство пользователя описывает специфические свойства языка UML. Тем не менее книга не задумывалась как исчерпывающее справочное руководство по UML. Эту функцию выполняет справочник Rumbaugh, Jacobson, Booch. The Unified Modeling Language Reference Manual. Second Edition (Addison-Wesley, 2005)1.

Предлагаемое вашему вниманию руководство описывает процесс разработки программных систем с использованием UML, однако не предоставляет полную информацию об этом процессе. Более подробные сведения приводятся в книге: Rumbaugh, Jacobson, Booch. Unified Software Development Process. Addison-Wesley,1999.

[1] Русскоязычное издание книги: Буч Г., Якобсон А., Рамбо Дж. UML. 2-е изд. – СПб.: Питер, 2006.

Наконец, в книге содержится множество рекомендаций и советов по использованию UML для решения часто возникающих задач моделирования, но моделированию как таковому она не учит. Подобный подход принят в руководствах по большинству языков программирования, разъясняющих, как применять язык, но не обучающих собственно программированию.

Для кого предназначена эта книга

Язык UML представляет интерес для любого специалиста, участвующего в процессе разработки, внедрения и сопровождения программного обеспечения. Данное руководство пользователя в первую очередь предназначено для разработчиков, создающих модели UML. В то же время оно будет полезно всем, кто читает эти модели в процессе анализа, создания, тестирования и выпуска версий программных систем. Хотя под такое определение подходит чуть ли не любой сотрудник организации, занимающейся разработкой программного обеспечения, книга *особенно* пригодится аналитикам и конечным пользователям, которые специфицируют требуемую структуру и поведение системы, архитекторам, которые проектируют системы в соответствии с этими требованиями, разработчикам, преобразующим проект в исполняемый код, специалистам по обеспечению качества, проверяющим и подтверждающим соответствие структуры и поведения системы заданным спецификациям, библиотекарям, которые создают каталоги компонентов, а также руководителям программ и проектов, которые борются с хаосом, осуществляют общее руководство, определяют направление работ и распределяют необходимые ресурсы.

Данная книга рассчитана на читателей, которые имеют хотя бы общее представление об объектно-ориентированных концепциях. Опыт работы с языками или методами объектно-ориентированного программирования не требуется, хотя и желателен.

Как работать с этой книгой

Тем, кто только начинает осваивать язык UML, лучше всего читать эту книгу последовательно. Особое внимание рекомендуется уделить второй главе, в которой излагается концептуальная модель языка. Содержание каждой главы опирается на материал предыдущей, что делает книгу удобной для последовательного чтения.

Опытные разработчики, желающие найти решение конкретных проблем, возникающих при моделировании, могут изучать это руководство в любой последовательности. Советуем обратить внимание на типичные приемы моделирования, представленные в каждой главе.

Организация книги и особенности изложения материала

Данное руководство пользователя содержит семь основных разделов:

- Часть I. Введение в процесс моделирования.
- Часть II. Основы структурного моделирования.
- Часть III. Расширенное структурное моделирование.
- Часть IV. Основы моделирования поведения.
- Часть V. Расширенное моделирование поведения.
- Часть VI. Моделирование архитектуры.
- Часть VII. Итоги.

Кроме того, в книгу включены два приложения: обзор применяемой в языке UML нотации и обзор технологии Rational Unified Process, а также глоссарий, содержащий наиболее распространенные термины.

Каждая глава посвящена рассмотрению какой-то конкретной возможности UML и, как правило, состоит из следующих четырех разделов:

1. Введение.
2. Термины и понятия.
3. Типичные приемы моделирования.
4. Советы и подсказки.

В начале каждой главы приводится список обсуждаемых в ней тем.

В разделе «Типичные приемы моделирования» содержатся примеры решения задач, часто возникающих при моделировании. Для облегчения работы с руководством каждая задача приводится под отдельным заголовком.

Комментарии и советы по теме главы приводятся в разделах «На заметку».

Язык UML имеет широкие семантические возможности. По этой причине описание одной особенности может пересекаться с описанием другой. В таких случаях слева приводятся ссылки на перекрестные темы.

Серый цвет ссылки используется для объяснения модели, которая, в отличие от объяснения, всегда изображается черным. Фрагменты кода выделяются `моноширинным шрифтом`.

Благодарности

Авторы хотят выразить благодарность Брюсу Дугласу (Bruce Douglass),

Перу Кроллу (Per Kroll) и Хоакину Миллеру (Joaquin Miller) за их помощь в создании книги.

Краткая история UML

Первым объектно-ориентированным языком принято считать Simula-67, разработанный Далем и Нигардом в Норвегии в 1967 году. Этот язык мало кто взял на вооружение, но его концепция во многом способствовала появлению других языков. Язык Smalltalk получил широкое распространение в начале 1980-х годов, а в конце того же десятилетия за ним последовали другие объектно-ориентированные языки, такие как Objective C, C++ и Eiffel. Объектно-ориентированные языки моделирования появились в 1980-х годах, когда исследователи, поставленные перед необходимостью учитывать новые возможности объектно-ориентированных языков программирования и требования, предъявляемые все более сложными приложениями, вынуждены были начать разработку различных альтернативных подходов к анализу и проектированию. В период с 1989 по 1994 год число объектно-ориентированных методов возросло с десяти более чем до пятидесяти. Тем не менее многие пользователи испытывали затруднения при выборе языка моделирования, который полностью отвечал бы их потребностям, что послужило причиной так называемой «войны методов». В результате появилось новое поколение методов, среди которых особое значение приобрели метод Буча, OOSE (Object-Oriented Software Engineering), разработанный Якобсоном, и OMT (Object Modeling Technique), разработанный Рамбо. Кроме того, следует упомянуть методы Fusion, Shlaer-Mellor и Coad-Yourdon. Каждый из этих методов можно было считать целостным и законченным, хотя любой из них имел не только сильные, но и слабые стороны. Выразительные возможности метода Буча были особенно важны на этапах проектирования и конструирования системы. OOSE был великолепно приспособлен для анализа и формулирования требований, а также для высокоуровневого проектирования. OMT оказался особенно полезным для анализа и разработки информационных систем.

Критическая масса новых идей начала накапливаться к середине 1990-х годов, когда Гради Буч (компания Rational Software Corporation), Джеймс Рамбо (General Electric), Ивар Якобсон (Objectory) и другие предприняли попытку объединить свои методы, уже получившие мировое признание как наиболее перспективные в области объектно-ориентированных методов. Являясь основными авторами методов Буча, OOSE и OMT, мы попытались создать новый, унифицированный язык моделирования и руководствовались при этом тремя соображениями. Во-первых, все три метода независимо от желания разработчиков уже развивались во встречном направлении. Разумно было продолжить это движение вместе, а не по отдельности, что помогло бы в будущем устранить нежелательные различия и, как следствие, неудобства для пользователей. Во-вторых, унифицировав

методы, проще было привнести стабильность на рынок инструментов объектно-ориентированного моделирования, что дало бы возможность положить в основу всех проектов единый зрелый язык, а создателям инструментальных средств позволило бы сосредоточиться на более продуктивной деятельности. Наконец, следовало полагать, что подобное сотрудничество приведет к усовершенствованию всех трех методов и обеспечит решение задач, для которых любой из них, взятый в отдельности, был не слишком пригоден.

Начав унификацию, мы поставили перед собой три главные цели:

1. Моделировать системы целиком, от концепции до исполняемых компонентов, с помощью объектно-ориентированных методов;
2. Решить проблему масштабируемости, которая присуща сложным, критически важным системам;
3. Создать язык моделирования, который может использоваться не только людьми, но и компьютерами.

Создание языка для объектно-ориентированного анализа и проектирования не слишком отличается от разработки языка программирования. Во-первых, требовалось ограничить задачу. Следует ли включать в язык возможность спецификации требований? Должен ли язык позволять визуальное программирование? Во-вторых, необходимо было найти точку равновесия между выразительной мощью и простотой. Слишком простой язык ограничил бы круг решаемых с его помощью задач, а слишком сложный мог ошеломить неискушенного разработчика. Кроме того, при объединении существующих методов следовало учитывать наличие продуктов, разработанных с их помощью. Внесение слишком большого числа изменений могло бы оттолкнуть уже имеющихся пользователей, а авторы, сопротивляясь развитию языка, потеряли бы возможность привлекать новых пользователей и делать язык более простым и удобным для применения. Создавая UML, мы старались найти оптимальное решение этих проблем.

Официальное создание UML началось в октябре 1994 года, когда Рамбо перешел в компанию Rational Software, где работал Буч. Первоначальной целью было объединение методов Буча и OMT. Первая пробная версия 0.8 Унифицированного метода (Unified Method), как его тогда называли, появилась в октябре 1995 года. Приблизительно в то же время в компанию Rational перешел Якобсон, и проект UML был расширен с целью включить в него OOSE. В результате наших совместных усилий в июне 1996 года вышла версия 0.9 языка UML. На протяжении всего года создатели занимались сбором отзывов от основных компаний, работающих в области создания программного обеспечения. За это время стало ясно, что большинство таких компаний сочло UML языком, имеющим стратегическое значение для их бизнеса.

В результате был создан консорциум UML, в который вошли организации, изъявившие желание предоставить ресурсы для работы, направленной на создание полной спецификации UML. Версия 1.0 языка появилась в результате совместных усилий компаний Digital Equipment Corporation, Hewlett-Packard, I-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments и Unisys. UML 1.0 оказался хорошо определенным, выразительным, мощным языком, применимым для решения большого количества разнообразных задач. В январе 1997 года он был предоставлен в качестве проекта стандартного языка моделирования консорциуму OMG (Object Management Group).

Между январем и июлем 1997 года консорциум UML расширился – в него вошли почти все компании, откликнувшиеся на призыв OMG, а именно: Andersen Consulting, Ericsson, Object Time Limited, Platinum Technology, Ptech, Reich Technologies, Softeam, Sterling Software и Taskon. Чтобы формализовать спецификации UML и координировать работу с другими группами, занимающимися стандартизацией, под руководством Криса Кобрина (Cris Kobryn) из компании MCI Systemhouse и Эда Эйхолта (Ed Eykholt) из Rational была организована рабочая группа. Пересмотренная версия UML (1.1) была представлена на рассмотрение OMG в июле 1997 года. В сентябре версия была утверждена на заседаниях группы по анализу и проектированию и Комитета по архитектуре OMG, а 14 ноября 1997 года была утверждена в качестве стандарта всеми участниками консорциума OMG.

В течение нескольких лет специальная рабочая группа OMG (OMG Revision Task Force) поддерживала продвижение проекта UML. Были созданы версии 1.3, 1.4 и 1.5. За 2000–2003 годы новая, расширенная группа участников проекта создала модернизированную версию UML 2.0. Эта версия рассматривалась в течение года рабочей группой Finalization Task Force (FTF) во главе с Брэном Сэликом (Bran Selic) из IBM, а в начале 2005 года члены OMG утвердили официальную версию UML 2.0. UML 2.0 – это последний релиз UML, который включает в себя большое количество дополнительных возможностей. Много изменений было сделано благодаря опыту использования предыдущих версий. Текущую спецификациюUML вы найдете на Web-сайте OMG www.omg.org.

UML – это результат работы множества специалистов и осмысления опыта их предыдущих разработок. Подсчитать источники, которые использовались при подготовке этого проекта, было бы огромным научным трудом. Но сложнее всего было бы установить все предшествующие наработки, в большей или меньшей степени оказавшие влияние на UML. С учетом всех научных исследований и рабочей практики можно сказать, что UML – это верхушка «айсберга», который вобрал в себя весь предыдущий опыт.

Часть I

Введение в процесс моделирования

Глава 1. Зачем мы моделируем

В этой главе:

- Значение моделирования
- Четыре принципа моделирования
- Наиболее важные модели программных систем
- Объектно-ориентированное моделирование

Компания, занимающаяся производством программного обеспечения, может преуспевать только в том случае, если выпускаемая ею продукция отличается высоким качеством и разработана в соответствии с запросами пользователей. Фирма, которая способна выпускать такую продукцию своевременно и регулярно, при максимально полном и эффективном использовании всех имеющихся человеческих и материальных ресурсов будет стабильно процветать.

Из сказанного следует, что основным продуктом такой компании является первоклассное программное обеспечение, удовлетворяющее повседневные нужды пользователей. Все остальное – прекрасные документы, встречи на высшем уровне, великолепные лозунги и даже Пулитцеровская премия за идеальные строки исходного кода – вторично по сравнению с этой основной задачей.

К сожалению, во многих организациях путают понятия «вторичный» и «несущественный». Нельзя забывать, что для разработки эффективной программы, которая соответствует своему предполагаемому назначению, необходимо постоянно встречаться и работать с пользователем, чтобы выяснить реальные требования к вашей системе. Если вы хотите создать качественное программное обеспечение, вам необходимо разработать прочное архитектурное основание проекта, открытое к возможным усовершенствованиям. Для быстрой и эффективной разработки программного продукта с минимальным браком требуется привлечь рабочую силу, выбрать правильные инструменты и определить верное направление работы. Чтобы справиться с поставленной задачей, принимая во внимание затраты на жизненный цикл системы, необходимо, чтобы процесс разработки приложения был тщательно продуман и мог

быть адаптирован к изменяющимся потребностям вашего бизнеса и технологии.

Центральным элементом деятельности, ведущей к созданию первоклассного программного обеспечения, является моделирование. Модели позволяют нам наглядно продемонстрировать желаемую структуру и поведение системы. Они также необходимы для визуализации и управления ее архитектурой. Модели помогают добиться лучшего понимания создаваемой нами системы, что зачастую приводит к ее упрощению и обеспечивает возможность повторного использования. Наконец, модели нужны для минимизации риска.

Значение моделирования

Если вы хотите соорудить собачью будку, то можете приступить к работе, имея в наличии лишь кучу досок, горсть гвоздей, молоток, плоскогубцы и рулетку. Несколько часов работы после небольшого предварительного планирования – и вы, надо полагать, сколотите вполне приемлемую будку, причем, скорее всего, без посторонней помощи. Если будка получится достаточно большой и не будет сильно протекать, собака останется довольна. В крайнем случае никогда не поздно начать все сначала – или приобрести менее капризного пса.

Если вам надо построить дом для своей семьи, вы, конечно, можете воспользоваться тем же набором инструментов, но времени на это уйдет значительно больше, и ваши домочадцы, надо полагать, окажутся более требовательными, чем собака. Если у вас нет особого опыта в области строительства, лучше тщательно все продумать перед тем, как забить первый гвоздь. Стоит, по меньшей мере, сделать хотя бы несколько эскизов будущей постройки. Без сомнения, вам нужно качественное жилье, удовлетворяющее запросам вашей семьи и не нарушающее местных строительных норм и правил, – а значит, придется сделать кое-какие чертежи с учетом назначения каждой комнаты и таких деталей, как освещение, отопление и водопровод. На основании этих планов вы сможете правильно рассчитать необходимое для работы время и выбрать подходящие стройматериалы. В принципе можно построить дом и самому, но гораздо выгоднее прибегнуть к помощи других людей, нанимая их для выполнения ключевых работ, или хотя бы купить готовые детали. Коль скоро вы следуете плану и укладываетесь в смету, ваша семья будет довольна. Если же что-то не сладится, вряд ли стоит менять семью – лучше своевременно учесть пожелания родственников.

Задавшись целью построить небоскреб для офиса, было бы совсем неразумно браться за дело, имея в распоряжении груду досок и молоток. Поскольку в этом случае, скорее всего, будут привлекаться

многочисленные капиталовложения, инвесторы потребуют, чтобы вы учли их пожелания относительно размера, стиля и формы строения. Кстати, они могут изменить свое решение уже после того, как вы приступите к строительству. Так как любой просчет вам обойдется слишком дорого, значение планирования многократно возрастает. По-видимому, вы окажетесь членом большого коллектива проектировщиков и застройщиков, и для успешного взаимодействия вам потребуется выполнять множество разнообразных чертежей и моделей здания.

Подобрав нужных людей, подходящие инструменты и приступив к реализации намеченного плана, вы с большой вероятностью сумеете построить здание, удовлетворяющее всем требованиям будущих обитателей. Если вы и дальше собираетесь возводить дома, то придется искать компромисс между пожеланиями заказчика и возможностями современной технологии строительства, а к рабочему коллективу относиться с заботой, не рискуя людьми и не требуя работы, отнимающей всех сил без остатка.

Хотя это и кажется комичным, многие компании, разрабатывающие программное обеспечение, хотят создать небоскреб, в то время как их подход к делу очень напоминает строительство собачьей будки.

Конечно, иногда вам может просто повезти. Если нужные люди оказались в нужном месте в нужное время и расположение планет благоприятствует новым начинаниям, то, возможно, вам посчастливится создать продукт, который подойдет потребителям. Но, скорее всего, нанять подходящих работников не удастся (все они уже заняты), благоприятный момент вы упустите (вчера было бы лучше), а расположение планет будет крайне неблагоприятным (этот факт вообще неподвластен вашему контролю). Сталкиваясь с возрастающими требованиями к скорости разработки программного обеспечения, коллективы разработчиков часто ограничиваются тем единственным, что они по-настоящему умеют делать – написанием новых строк кода. Героические усилия, затрачиваемые на программирование, стали легендой в этой отрасли, и кажется, что единственным ответом на трудности в разработке программного обеспечения послужит еще более интенсивная работа. Однако написанный код вовсе не обязательно поможет решить проблему, а проект может быть настолько грандиозным, что даже увеличение рабочего дня на несколько часов окажется недостаточным для его успешного завершения.

Если вы действительно хотите создать программный продукт, по масштабности замысла сопоставимый с жилым домом или небоскребом, то ваша задача не сводится к написанию большого объема кода. На самом деле проблема в том, чтобы написать *правильный* код *минимального* объема. При таком подходе разработка качественного программного обеспечения сводится к вопросам выбора архитектуры, подходящего инструментария и средств управления процессом. Кстати, нужно иметь в виду, что многие проекты, задуманные «по принципу будки», быстро вырастают до размеров небоскреба, становясь жертвой собственного успеха. Если такой рост не был учтен в архитектуре приложения, технологическом процессе или при выборе инструментария, то неизбежно наступает время, когда выросшая до размеров огромного дома будка обрушится под тяжестью собственного веса. Но если разрушится будка, то это лишь разозлит вашу собаку, а если рухнет небоскреб, это затронет материальные интересы его арендаторов.

Неудачные проекты терпят крах в силу самых разных причин, а вот успешные, как правило, имеют много общего. Хотя успех программного проекта обеспечивается множеством разных составляющих, одной из главных является применение моделирования.

Моделирование – это устоявшаяся и повсеместно принятая инженерная методика. Мы строим архитектурные модели зданий, чтобы помочь их будущим обитателям во всех деталях представить готовый объект. Иногда применяют даже математическое моделирование зданий, чтобы учесть влияние сильного ветра или землетрясения.

Моделирование применяется не только в строительстве. Вряд ли вы сумеете наладить выпуск новых самолетов или автомобилей, не испытав свой проект на моделях: от компьютерных моделей и чертежей до физических моделей в аэродинамической трубе, а затем и полномасштабных прототипов. Электрические приборы от микропроцессоров до телефонных коммутаторов также требуют моделирования для лучшего понимания системы и организации общения разработчиков друг с другом. Даже в кинематографии успех фильма невозможен без предварительно написанного сценария (тоже своеобразная форма модели!) В социологии, экономике или менеджменте мы также прибегаем к моделированию, которое позволяет проверить наши теории и испытать новые идеи с минимальным риском и затратами.

Итак, что же такое модель? Попросту говоря, *модель – это упрощенное представление реальности*.

Модель – это чертеж системы: в нее может входить как детальный план, так и более абстрактное представление системы «с высоты птичьего полета». Хорошая модель всегда включает элементы, которые существенно влияют на результат, и не включает те, которые малозначимы на данном уровне абстракции. Каждая система может быть описана с разных точек зрения, для чего используются разные модели, каждая из которых, следовательно, является семантически

замкнутой абстракцией системы. Модель может быть структурной, подчеркивающей организацию системы, или же поведенческой, отражающей ее динамику.

Зачем мы моделируем? Для этого есть одна фундаментальная причина.

Мы строим модель для того, чтобы лучше понимать разрабатываемую систему.

Моделирование позволяет решить четыре различные задачи (см. главу 2):

1. Визуализировать систему в ее текущем или желательном для нас состоянии;
2. Описать структуру или поведение системы;
3. Получить шаблон, позволяющий сконструировать систему;
4. Документировать принимаемые решения, используя полученные модели.

Моделирование предназначено не только для создания больших систем. Даже программный эквивалент собачьей будки выиграет от его применения. Чем больше и сложнее система, тем большее значение приобретает моделирование при ее разработке. Дело в том, что *мы строим модели сложных систем, поскольку иначе такие системы невозможно воспринять как единое целое.*

Человеческое восприятие сложных сущностей ограничено. Моделируя, мы сужаем проблему, акцентируем внимание в каждый данный момент только на одном ее аспекте. По сути, этот подход есть не что иное, как принцип «разделяй и властвуй», который Эдсгер Дейкстра (Edsger Dijkstra) провозгласил много лет назад: сложную задачу легче решить, если разделить ее на несколько меньших. Кроме того, моделирование усиливает возможности человеческого интеллекта, поскольку правильно выбранная модель позволяет создавать проекты на более высоких уровнях абстракции.

Сказать, что моделирование имеет смысл, – еще не значит, что оно абсолютно необходимо. И действительно, многие исследования показывают, что в большинстве компаний, разрабатывающих программное обеспечение, моделирование применяется редко или же не применяется вообще. Чем проще проект, тем менее вероятно, что в нем будет использовано формальное моделирование.

Ключевое слово здесь – «формальное». На практике даже при реализации простейшего проекта разработчики в той или иной мере применяют моделирование, хотя бы неформально. Для визуализации части системы ее проектировщик может нарисовать что-то на доске или клочке бумаги. Коллектив разработчиков может применять CRC-карточки, чтобы проработать сценарий или конструкцию механизма. Ничего плохого в таких моделях нет. Если они

работают, то их существование оправдано. Но такие неформальные модели часто создаются для однократного применения и не обеспечивают общего языка, который был бы понятен другим участникам проекта. В строительстве существует общий, понятный для всех язык чертежей, имеются общие языки в электротехнике и математическом моделировании. Организация разработчиков также может извлечь выгоду из применения общего языка моделирования программного обеспечения.

От моделирования может выиграть любой проект. Даже при создании одноразовых программ, когда зачастую бывает полезно выбрать неподходящий код из-за преимущества в скорости разработки, которое дают языки визуального программирования, моделирование поможет коллективу разработчиков лучше представить план системы, а значит, выполнить проект быстрее и создать именно то, что подразумевал первоначальный замысел. Чем сложнее проект, тем более вероятно, что из-за отсутствия моделирования он свернется раньше времени – или будет создано не то, что нужно. Все полезные и интересные системы с течением времени обычно усложняются. Пренебрегая моделированием в самом начале создания системы, вы, возможно, серьезно пожалеете об этом, когда будет уже слишком поздно.

Принципы моделирования

Моделирование имеет богатую историю во всех инженерных дисциплинах. Длительный опыт его использования позволил сформулировать четыре основных принципа.

Во-первых, *выбор модели оказывает определяющее влияние на подход к решению проблемы и на то, как будет выглядеть это решение.*

Иначе говоря, подходите к выбору модели вдумчиво. Правильно выбранная модель высветит самые коварные проблемы разработки и позволит проникнуть в самую суть задачи, что при ином подходе было бы просто невозможно. Неправильная модель заведет вас в тупик, поскольку внимание будет заостряться на несущественных вопросах.

Давайте отвлечемся на минутку от обсуждения программного обеспечения и предположим, что вам нужно решить задачу из области квантовой физики. Такие проблемы, как взаимодействие фотонов в пространстве-времени, могут потребовать чрезвычайно сложных математических расчетов. Но стоит изменить модель, забыв о математическом анализе, и вы тут же получите легко интерпретируемый результат. В данном случае речь идет о диаграммах Феймана, позволяющих графически представить чрезвычайно сложные проблемы. Рассмотрим еще одну область, где необходимо моделирование.

Допустим, вы проектируете здание и хотите знать, как оно будет себя вести при сильном ветре. Построив макет и испытав его в аэродинамической трубе, вы узнаете много интересного по этому поводу, хотя уменьшенная копия будет себя вести не так, как полно ем математической модели даст вам дополнительную информацию и, возможно, позволит проработать больше различных сценариев, чем при работе с физическим макетом. Тщательно изучив поведение здания на этих моделях, вы будете гораздо больше уверены в том, что смоделированная система будет вести себя в реальных условиях так, как было задумано.

Возвращаясь к проблеме создания программного обеспечения, можно сказать, что ваш взгляд на мир существенно зависит от выбираемой вами модели. Если вы смотрите на систему глазами разработчика баз данных, то основное внимание будете уделять моделям «сущность–связь», где поведение инкапсулировано в триггерах и хранимых процедурах. Аналитик, использующий структурный подход, скорее всего, создал бы модель, в центре которой находятся алгоритмы и передача данных от одного процесса к другому. Результатом труда разработчика, пользующегося объектно-ориентированным методом, будет система, архитектура которой основана на множестве классов и образцах взаимодействия, определяющих, как эти классы действуют совместно. Любой из этих вариантов может оказаться подходящим для данного приложения и методики разработки, хотя опыт подсказывает, что объектно-ориентированная точка зрения более эффективна при создании гибких архитектур, даже если система должна будет работать с большими базами данных или производить сложные математические расчеты. При этом надо учитывать, что различные точки зрения на мир приводят к созданию различных систем, со своими преимуществами и недостатками.

Второй принцип формулируется так: *каждая модель может быть представлена с различной степенью точности*.

При строительстве небоскреба может возникнуть необходимость показать его с высоты птичьего полета, например, чтобы с проектом могли ознакомиться инвесторы. В других случаях, наоборот, требуется детальное описание – допустим, чтобы показать какой-нибудь сложный изгиб трубы или необычный элемент конструкции.

То же происходит и при моделировании программного обеспечения. Иногда простая и быстро созданная модель программного интерфейса – самый подходящий вариант. В других случаях приходится работать на уровне битов (например, когда вы специфицируете межсистемные интерфейсы или боретесь с узкими местами в сети). В любом случае лучшей моделью будет та, которая позволяет выбрать уровень детализации в зависимости от того, кто и с какой целью на нее смотрит. Для аналитика или конечного пользователя

наибольший интерес представляет вопрос «что», а для разработчика – вопрос «как». В обоих случаях необходима возможность рассматривать систему на разных уровнях детализации в разное время.

Третий принцип: *лучшие модели – те, что ближе к реальности*.

Физическая модель здания, которая ведет себя не так, как изготовленная из реальных материалов, имеет лишь ограниченную ценность. Математическая модель самолета, для которой предполагаются идеальные условия работы и безупречная сборка, может и не обладать некоторыми характеристиками, присущими настоящему изделию, что в ряде случаев приводит к фатальным последствиям. Лучше всего, если ваши модели будут во всем соотноситься с реальностью, а там, где связь ослабевает, должно быть понятно, в чем заключается различие и что из этого следует. Поскольку модель всегда упрощает реальность, задача в том, чтобы это упрощение не повлекло за собой какие-то существенные потери.

Возвращаясь к программному обеспечению, можно сказать, что «ахиллесова пята» структурного анализа – несоответствие принятой в нем модели и модели системного проекта. Если этот разрыв не будет устранен, то поведение созданной системы с течением времени будет все больше и больше отличаться от задуманного. При объектно-ориентированном подходе можно объединить все почти независимые представления системы в единое семантическое целое.

Четвертый принцип заключается в том, что *нельзя ограничиваться созданием только одной модели. Наилучший подход при разработке любой нетривиальной системы – использовать совокупность нескольких моделей, почти независимых друг от друга*.

Если вы конструируете здание, никакой отдельный комплект чертежей не поможет вам прояснить все детали до конца. Понадобятся как минимум поэтажные планы, виды в разрезе, схемы электропроводки, центрального отопления и водопровода.

В формулировке четвертого принципа ключевое определение – «почти независимые». Оно означает, что модели могут создаваться и изучаться по отдельности, но вместе с тем остаются взаимосвязанными. Например, можно изучать только схемы электропроводки проектируемого здания, но при этом наложить их на поэтажный план пола и даже рассмотреть совместно с прокладкой труб на схеме водоснабжения.

Такой подход верен и в объектно-ориентированных программных системах. Для понимания архитектуры подобной системы требуется несколько взаимодополняющих представлений: представление с точки зрения вариантов использования (чтобы выявить требования к системе), с точки зрения проектирования (чтобы построить словарь предметной области и области решения), с точки зрения взаимодействий (чтобы смоделировать взаимодействия

между частями системы, системой в целом и средой ее функционирования), с точки зрения реализации (позволяющее рассмотреть физическую реализацию системы) и с точки зрения размещения (помогающее сосредоточиться на вопросах системного проектирования). Каждое из перечисленных представлений имеет множество структурных и поведенческих аспектов, которые в своей совокупности составляют детальный чертеж программной системы.

Пять представлений архитектуры обсуждаются в главе 2.

В зависимости от природы системы некоторые модели могут быть важнее других. Так, при создании систем для обработки больших объемов данных более важны статические модели. В приложениях, ориентированных на интерактивную работу пользователя, на первый план выходят статические и динамические представления вариантов использования. В системах реального времени наиболее существенными будут представления с точки зрения динамических процессов. Наконец, в распределенных системах, таких как Web-приложения, основное внимание нужно уделять моделям реализации и размещения.

Объектное моделирование

Инженеры-строители создают множество видов моделей. Чаще всего это структурные модели, позволяющие визуализировать и специфицировать части системы и то, как они соотносятся друг с другом. Иногда создаются также динамические модели – например, если требуется изучить поведение конструкции при землетрясении. Эти два типа различаются по организации и по тому, на что в первую очередь обращается внимание при проектировании.

При разработке программного обеспечения тоже существует несколько подходов к моделированию. Важнейшие из них – алгоритмический и объектно-ориентированный.

Алгоритмический метод представляет традиционный подход к созданию программного обеспечения. Основным строительным блоком является процедура или функция, а внимание уделяется прежде всего вопросам передачи управления и вопросам декомпозиции больших алгоритмов на меньшие. Ничего плохого в этом нет, если не считать того, что системы не слишком легко адаптируются. При изменении требований или увеличении размера приложения (что происходит нередко) сопровождать их становится сложнее.

Наиболее современный подход к разработке программного обеспечения – объектно-ориентированный. Здесь в качестве основного строительного блока выступает объект или класс. В самом общем смысле *объект* – это сущность, обычно извлекаемая из словаря предметной области или решения, а *класс* – описание множества однотипных объектов. Каждый объект обладает идентичностью

(его можно поименовать или как-то по-другому отличить от прочих объектов), состоянием (обычно с объектом связаны некоторые данные) и поведением (с ним можно что-то делать или он сам может что-то делать с другими объектами).

В качестве примера давайте рассмотрим простую трехуровневую архитектуру биллинговой системы, состоящую из интерфейса пользователя, бизнес-логики (промежуточного слоя) и базы данных. Пользовательский интерфейс состоит из конкретных объектов: кнопок, меню и диалоговых окон. База данных также состоит из конкретных объектов, а именно таблиц, представляющих сущности предметной области: клиентов, продукты и заказы. Программы промежуточного слоя включают такие объекты, как транзакции и бизнес-правила, а кроме того, более абстрактные представления сущностей предметной области (клиентов, продуктов и заказов).

Объектно-ориентированный подход к разработке программного обеспечения сейчас наиболее широко используется потому, что он продемонстрировал свою полезность при построении систем любого размера и сложности в самых разных областях. Кроме того, большинство современных языков программирования, инструментальных средств и операционных систем являются в той или иной мере объектно-ориентированными, а это дает веские основания судить о мире в терминах объектов. Объектно-ориентированные методы разработки легли в основу идеологии сборки систем из отдельных компонентов (в качестве примеров можно назвать такие технологии, как J2EE и .NET).

Если вы приняли объектно-ориентированный взгляд на мир, вам придется ответить на ряд вопросов. Какая структура должна быть у хорошей объектно-ориентированной архитектуры? Какие артефакты должны быть созданы в процессе работы над проектом? Кто должен создавать их? И наконец, как оценить результат?

Компоненты обсуждаются в главе 2.

Визуализация, специфицирование, конструирование и документирование объектно-ориентированных систем – это и есть назначение языка UML.

Глава 2. Введение в UML

В этой главе:

- Обзор UML
- Три шага к пониманию UML
- Архитектура программного обеспечения
- Процесс разработки программного обеспечения

Итак, унифицированный язык моделирования (Unified Modeling Language – UML) – это стандартный инструмент для разработки «чертежей» программного обеспечения. Его можно использовать для визуализации, спецификации, конструирования и документирования артефактов программных систем.

UML подходит для моделирования любых систем – от информационных систем масштаба предприятия до распределенных Web-приложений и даже встроенных систем реального времени. Это очень выразительный язык, предназначенный для представления системы со всех точек зрения, относящихся к ее разработке и внедрению. Несмотря на богатство выразительных средств, UML прост для понимания и применения. Изучение эффективного использования UML начинается с формирования концептуальной модели языка, и в этой связи необходимо ознакомиться с *тремя основными элементами*: базовыми строительными блоками UML, правилами, определяющими, как эти блоки могут сочетаться между собой, а также некоторыми общими механизмами языка.

UML – всего лишь язык, и как таковой представляет только одну из составляющих процесса разработки программного обеспечения. Хотя UML не зависит от моделируемых процессов, лучше всего применять его в тех случаях, когда процесс моделирования основан на применении вариантов использования, сконцентрирован на архитектуре системы, является итеративным и пошаговым.

Обзор UML

UML – это язык для *визуализации*, *специфицирования*, *конструирования* и *документирования* артефактов программных систем.

Язык представляет словарь и правила комбинирования входящих в него слов в целях коммуникации. *Язык моделирования* – это язык, словарь и правила которого сосредоточены на концептуальном и физическом представлении системы. UML – стандартное средство представления «чертежей» программного обеспечения.

Основные принципы моделирования обсуждаются в главе 1.

Моделирование необходимо для понимания системы. При этом ни одна модель не является абсолютно достаточной. Напротив, чтобы понять большинство систем, кроме самых тривиальных, часто требуется множество взаимосвязанных моделей. В отношении программных систем это означает, что необходим язык, средствами которого можно описать архитектуру системы с различных точек зрения, причем на протяжении всего жизненного цикла ее разработки.

Словарь и правила такого языка, как UML, говорят о том, *как* создавать и читать хорошо согласованные модели, но не говорит о том, *какие* именно модели в каких случаях требуется создавать. Это задача всего процесса разработки программного обеспечения. Хорошо организованный процесс должен сам подсказать, какие потребуются рабочие продукты, какие ресурсы понадобятся для их создания и управления ими, как их использовать для оценки выполненной работы и управления проектом в целом.

UML — язык визуализации

С точки зрения многих программистов, промежуток времени между размышлениями о реализации проекта и их изложением в коде стремится к нулю. Вы думаете – значит, вы кодируете. И действительно, некоторые вещи лучше всего выражаются непосредственно в коде на языке программирования, потому что текст программ – самый прямой и короткий путь написания выражений и алгоритмов. Но и в этих случаях программист на самом деле занимается моделированием, хотя и делает это мысленно. Он может даже делать наброски некоторых идей – на доске или салфетке. Однако при этом возникают некоторые проблемы. Во-первых, обсуждение таких концептуальных моделей с другими участниками разработки чревато ошибками и непониманием, если только все участники дискуссии не говорят на одном языке. Как правило, при разработке проектов предприятиям приходится создавать в этих целях свои собственные языки, и вам трудно понять, о чем идет речь, если вы посторонний или новичок в группе. Во-вторых, некоторые вещи, касающиеся программных систем, трудно выразить, пытаясь строить модели лишь средствами текстовых языков программирования. Например, назначение иерархии классов можно понять, внимательно изучив код всех классов в иерархии, но воспринять всю структуру целиком не получится. Аналогично, изучая код, можно исследовать физическое представление

и распределение объектов в Web-ориентированной системе, но нельзя сразу «схватить» его целиком. В-третьих, если разработчик, который писал этот код, никогда не воплощал в нем модели, существовавшие в его голове, то информация о них может быть потеряна навсегда и в лучшем случае частично восстановлена на основе существующей реализации, когда этот разработчик перейдет на другую работу.

Описание моделей на UML позволяет решить третью проблему: явная модель облегчает общение.

Некоторые вещи лучше моделировать в тексте, другие – графически. В действительности во всех интересных системах существуют структуры, которые невозможно выразить на языке программирования. UML – графический язык, позволяющий решить вторую из описанных выше проблем.

UML – нечто большее, чем просто набор графических символов. Каждый из этих символов имеет четко определенную семантику. И это значит, что один разработчик может описать модель на UML, а другой разработчик и даже инструментальное средство – однозначно интерпретировать ее. Это решает первую из упомянутых проблем.

UML — язык специфицирования

В данном контексте *специфицирование* – это построение точных, недвусмысленных и полных моделей. В частности, UML позволяет специфицировать все важные решения, касающиеся анализа, дизайна и реализации, принимаемые в процессе разработки и внедрения программных систем.

UML — язык конструирования

UML не является визуальным языком программирования, но его модели могут быть непосредственно ассоциированы с различными языками программирования. А это значит, что существует возможность отобразить UML-модель на такой язык, как Java, C++ или Visual Basic, а при необходимости даже на таблицы реляционной базы данных либо объекты, хранящиеся в объектно-ориентированной базе данных. Те вещи, которые проще выразить графически, выражаются на UML, а те, что легче выразить в виде текста, – на языке программирования.

Моделирование структуры систем обсуждается в частях II и III.

Отображение модели на язык программирования позволяет осуществить *прямое проектирование* (forward engineering) – генерацию кода на языке программирования из модели UML. Обратное также возможно: вы можете восстановить модель UML на основе существующей реализации. В *обратном проектировании* (reverse engineering) нет никакой магии. Если только вы не закодировали информацию в реализации, она теряется при переходе от модели к коду.

Поэтому обратное проектирование, выполняемое инструментальными средствами, все же требует определенного вмешательства человека. Комбинация этих двух путей – прямого и обратного проектирования – обеспечивает возможность работы как с графическим, так и с текстовым представлениями; при этом обеспечивается согласованность между ними.

В дополнение к прямому отображению UML благодаря своей выразительности и однозначности позволяет непосредственно исполнять модели, имитируя поведение проектируемых систем, а также управляя действующими системами.

UML — язык документирования

Успешные компании, специализирующиеся на программном обеспечении, помимо исполняемого кода производят и другие продукты, включая следующие (но не ограничиваясь ими):

- требования;
- архитектуру;
- проектные решения (дизайн);
- исходный код;
- проектные планы;
- тесты;
- прототипы;
- релизы (версии).

В зависимости от уровня культуры разработки, принятой в компании, некоторые из этих продуктов выражаются более формально, чем другие. Перечисленные продукты – это не только поставляемые составные части проектов; они необходимы для управления, оценки результатов и взаимодействия в процессе разработки системы и после ее внедрения.

UML предназначен для документирования архитектуры системы и всех ее деталей. Кроме того, это язык для выражения требований к системе и описания тестов. И наконец, он подходит для моделирования работ на этапе проектирования и управления версиями.

Где может использоваться UML?

UML прежде всего предназначен для моделирования и разработки программных систем. Наиболее эффективно его применение в следующих областях:

- корпоративные информационные системы;
- банковские и финансовые услуги;
- телекоммуникации;
- транспорт;

- оборона, авиация и космонавтика;
- розничная торговля;
- медицинская электроника;
- наука;
- распределенные Web-сервисы.

Но сфера применения UML не ограничена моделированием программного обеспечения. Его выразительность позволяет вести работу и над непрограммными системами – в частности, продумывать документооборот юридической системы, структуру и функционирование системы здравоохранения, системы управления воздушным движением, а также проектировать аппаратные средства.

Концептуальная модель UML

Чтобы понять UML, вам необходимо сформировать концептуальную модель языка, а это требует изучения трех основных элементов: строительных блоков языка, правил, определяющих их сочетания, и некоторых общих для всего языка механизмов. Лишь усвоив эти идеи, вы сможете читать UML-модели и создавать их самостоятельно. По мере приобретения опыта в использовании UML вы сможете строить концептуальные модели, используя более развитые языковые средства.

Строительные блоки UML

Словарь UML включает три вида строительных блоков:

1. Сущности.
2. Связи.
3. Диаграммы.

Сущности (things) – это абстракции, которые являются основными элементами модели, *связи* (relationships) соединяют их между собой, а диаграммы (diagrams) группируют представляющие интерес наборы сущностей.

Есть четыре вида сущностей UML:

1. Структурные.
2. Поведенческие.
3. Группирующие.
4. Аннотирующие.

Все они представляют собой базовые объектно-ориентированные строительные блоки моделей UML. Вы используете их для описания хорошо согласованных моделей.

Структурные сущности – «имена существительные» в моделях UML. Это в основном статические части модели, представляющие

либо концептуальные, либо физические элементы. В совокупности структурные сущности называются *классификаторами* (classifiers).

Классы обсуждаются в главах 4 и 9.

Класс (class) – это описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой. Класс реализует один или несколько интерфейсов. Графически класс изображается в виде прямоугольника, обычно включающего имя, атрибуты и операции, как показано на рис. 2.1.

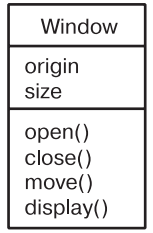


Рис. 2.1. Классы

Интерфейсы обсуждаются в главе 11.

Интерфейс (interface) – это набор операций, который специфицирует сервис (набор услуг) класса или компонента. Таким образом, интерфейс описывает видимое извне поведение элемента. Может представлять полное поведение класса или компонента либо только часть такого поведения. Определяет набор спецификаций операций (то есть их сигнатуру), но никогда не определяет детали их реализации. Объявление интерфейса изображается как класс с ключевым словом «interface» над его именем; атрибуты несущественны, за исключением иногда показываемых констант. Интерфейс, однако, редко существует сам по себе. Интерфейс, представляемый классом для внешнего мира, изображается в виде маленького круга, соединенного линией с рамкой класса. Интерфейс, запрашиваемый классом от некоторого другого класса, представлен маленьким полукругом, соединенным с рамкой класса линией, как показано на рис. 2.2.

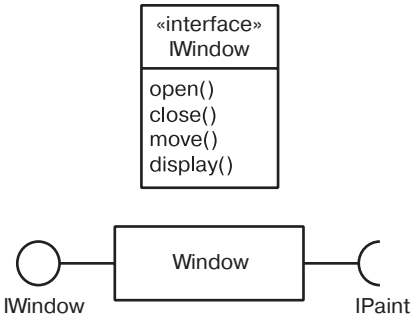


Рис. 2.2. Интерфейсы

Кооперации обсуждаются в главе 28.

Кооперация (collaboration) определяет взаимодействие и представляет собой совокупность ролей и других элементов, которые функционируют вместе, обеспечивая некоторое совместное поведение, представляющее нечто большее, чем сумма поведений отдельных элементов. Кооперации имеют как структурное, так и поведенческое измерения. Конкретный класс или объект может участвовать в нескольких кооперациях. Последние, таким образом, представляют собой реализацию *образцов* (patterns), составляющих систему. Кооперация изображается в виде эллипса, нарисованного пунктирной линией, иногда включающего в себя лишь ее имя, как на рис. 2.3.

Рис. 2.3. Кооперации

Варианты использования обсуждаются в главе 17.

Вариант использования (use case) – это описание последовательности действий, выполняемых системой и приносящих значимый результат конкретному *действующему лицу* (actor). Варианты использования применяются для структурирования поведенческих сущностей модели. Реализуются посредством коопераций. Графически вариант использования представлен эллипсом, нарисованным сплошной линией (обычно он включает в себя только имя, как показано на рис. 2.4).

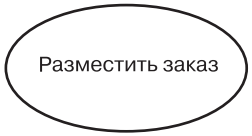


Рис. 2.4. Варианты использования

Оставшиеся три сущности – активные классы, компоненты и узлы (nodes) – все подобны классам; это говорит о том, что они также описывают наборы сущностей, разделяющих одни и те же атрибуты, операции, связи и семантику. Однако эти три понятия в достаточной степени различаются и необходимы для моделирования определенных аспектов объектно-ориентированных систем, поэтому нуждаются в специальном рассмотрении.

Активные классы обсуждаются в главе 23.

Активный класс – это класс, объекты которого являются владельцами одного или нескольких процессов или потоков (threads) и, таким образом, могут инициировать управляющие воздействия. Активный класс во всем подобен простому классу, за исключением того, что его объекты представляют собой элементы, поведение которых осуществляется параллельно с поведением других

элементов. Изображается как класс с двойными боковыми линиями; обычно включает в себя имя, атрибуты и операции, как показано на рис. 2.5.

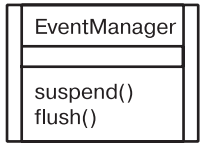


Рис. 2.5. Активные классы

Компоненты и внутренние структуры обсуждаются в главе 15.

Компонент – это модульная часть системы, которая скрывает свою реализацию за набором внешних интерфейсов. Компоненты системы, разделяющие общие интерфейсы, могут замещать друг друга, сохраняя при этом одинаковое логическое поведение. Реализация компонента может быть выражена объединением частей и коннекторов; при этом части могут включать в себя более мелкие компоненты. Графически компонент представлен как класс со специальной пиктограммой в правом верхнем углу (см. рис. 2.6).

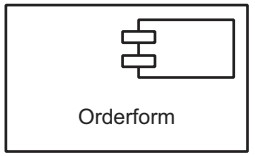


Рис. 2.6. Компоненты

Оставшиеся два элемента – артефакты и узлы – также различаются. Они представляют собой физические сущности, в отличие от предыдущих пяти, относящихся к сущностям логическим или концептуальным).

Артефакты обсуждаются в главе 26.

Артефакт (artifact) – это физическая и замещаемая часть системы, несущая физическую информацию («биты»). В системе вы можете встретить разные виды артефактов, таких как файлы исходного кода, исполняемые программы и скрипты. Обычно артефакт представляет собой физический пакет с исходным или исполняемым кодом. Изображается как прямоугольник, снабженный ключевым словом «artifact», расположенным над его именем (рис. 2.7).

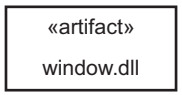


Рис. 2.7: Артефакты

Узлы описываются в главе 27.

Узел (node) – это физический элемент, который существует во время исполнения и представляет вычислительный ресурс, обычно имеющий по меньшей мере некоторую память и часто – вычислительные возможности. Набор компонентов может находиться на узле, а также мигрировать с одного узла на другой. Узел изображается в виде куба, обычно содержащего лишь его имя, как показано на рис. 2.8.

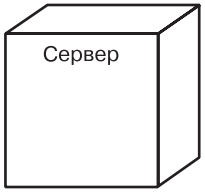


Рис. 2.8. Узлы

Все эти элементы – классы, интерфейсы, кооперации, варианты использования, активные классы, компоненты, артефакты и узлы – являются базовыми структурными сущностями, которые могут быть включены в UML-модель. Существуют также различные вариации: действующие лица (actors), сигналы и утилиты (разновидность классов), процессы и потоки (разновидности активных классов), приложения, документы, файлы, библиотеки, страницы и таблицы (разновидности артефактов).

Варианты использования, которые применяются для структурирования поведенческих сущностей в модели, обсуждаются в главе 17, взаимодействия (interactions) – в главе 16.

Поведенческие сущности – динамические части моделей UML. Это «глаголы» моделей, представляющие поведение во времени и пространстве. Всего существует три основных вида поведенческих сущностей.

Первый из них – *взаимодействие* (interaction) – представляет собой поведение, которое заключается в обмене сообщениями между наборами объектов или ролей в определенном контексте для достижения некоторой цели. Поведение совокупности объектов или индивидуальная операция могут быть выражены взаимодействием. Взаимодействие включает множество других элементов – таких как сообщения, действия (actions) и коннекторы (соединения между объектами). Сообщение изображается в виде линии со стрелкой, почти всегда сопровождаемой именем операции (см. рис. 2.9).

Рис. 2.9. Сообщения

Автоматы обсуждаются в главе 22.

Вторая из поведенческих сущностей – *автомат* (state machine) – представляет собой поведение, характеризуемое последовательностью состояний объекта, в которых он оказывается на протяжении своего жизненного цикла в ответ на события, вместе с его реакцией на эти события. Поведение индивидуального класса или кооперации классов может быть описано в терминах автомата. Автомат включает в себя множество других элементов: состояния, переходы (из одного состояния в другое), события (сущности, которые инициируют переходы), а также действия (реакции на переходы). Графически состояние представлено прямоугольником с закругленными углами, обычно с указанием имени и подсостояний, если таковые есть (см. рис. 2.10).

Рис. 2.10. Состояния

Третья из поведенческих сущностей – *деятельность* (activity) – специфицирует последовательность шагов процесса вычислений. Во взаимодействии внимание сосредоточено на наборе взаимодействующих объектов, в автомате – на жизненном цикле одного объекта; для деятельности же в центре внимания – последовательность шагов безотносительно к объектам, выполняющим каждый шаг. Отдельный шаг деятельности называется *действием* (action). Изображается оно в виде прямоугольника с закругленными углами, включающего имя, которое отражает его назначение. Состояния и действия различаются по контекстам.

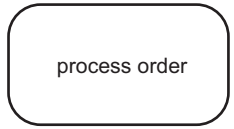


Рис. 2.11. Действия

Эти три элемента – взаимодействия, автоматы и деятельности – представляют собой базовые поведенческие сущности, которые вы можете включить в UML-модель. Семантически эти элементы обычно связаны с различными структурными элементами – в первую очередь, классами, кооперациями и объектами.

Группирующие сущности – организационная часть моделей UML. Это «ящики», по которым можно разложить модель. Главная из группирующих сущностей – пакет.

Пакеты обсуждаются в главе 12.

Пакет (package) – это механизм общего назначения для организации проектных решений, который упорядочивает конструкции реализации. Структурные сущности, поведенческие сущности и даже

другие группирующие сущности могут быть помещены в пакет. В отличие от компонентов (существующих только во время исполнения), пакеты полностью концептуальны, то есть существуют лишь на этапе разработки. Пакет изображается в виде папки с закладкой, обычно только с указанием имени, но иногда и содержимого (см. рис. 2.12).

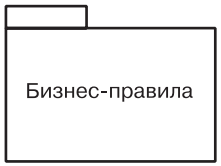


Рис. 2.12. Пакеты

Пакеты – основная группирующая сущность, с помощью которой вы можете организовать UML-модель. Существуют и такие вариации, как каркасы (frameworks), модели, подсистемы (разновидность пакетов).

Аннотирующие сущности – это поясняющие части UML-моделей, иными словами, комментарии, которые вы можете применить для описания, выделения и пояснения любого элемента модели. Главная из аннотирующих сущностей – *примечание* (note). Это простой символ, служащий для описания ограничений и комментариев, относящихся к элементу либо набору элементов. Графически представлен прямоугольником с загнутым углом; внутри помещается текстовый или графический комментарий (рис. 2.13).

Рис. 2.13. Примечания

Этот элемент – базовая аннотирующая сущность, которую вы можете включить в UML-модель. Обычно вы будете использовать его для снабжения диаграмм ограничениями или комментариями, которые лучше всего выражаются в виде формального или неформального текста. Существуют также разные вариации этого элемента, такие как требования, которые специфицируют некоторое желательное поведение с точки зрения, внешней по отношению к модели.

Существует четыре типа **связей в UML**:

1. Зависимость.
2. Ассоциация.
3. Обобщение.
4. Реализация.

Эти связи представляют собой базовые строительные блоки для описания отношений в UML, используемые для разработки хорошо согласованных моделей.

Зависимости обсуждаются в главах 5 и 10.

Первая из них – *зависимость* (dependency) – семантически представляет собой связь между двумя элементами модели, в которой изменение одного элемента (независимого) может привести к изменению семантики другого элемента (зависимого). Графически представлена пунктирной линией, иногда со стрелкой; может быть снабжена меткой (см. рис. 2.14).

Рис. 2.14. Зависимости

Ассоциации обсуждаются в главах 5 и 10.

Вторая, *ассоциация* (association), – это структурная связь между классами, которая описывает набор связей, существующих между объектами – экземплярами классов. *Агрегация* (aggregation) – особая разновидность ассоциации, представляющая структурную связь целого с его частями. Изображается сплошной линией, иногда со стрелкой; иногда снабжена меткой и часто содержит другие пометки, такие как мощность и конечные имена (см. рис. 2.15).

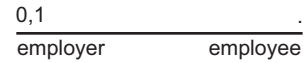


Рис. 2.15. Ассоциации

Обобщения рассматриваются в главах 5 и 10.

Третья связь – *обобщение* (generalization) – выражает специализацию или обобщение, в котором специализированный элемент (потомок) строится по спецификациям обобщенного элемента (родителя). Потомок разделяет структуру и поведение родителя. Графически обобщение представлено в виде сплошной линии с пустой стрелкой, указывающей на родителя (см. рис. 12.16).

Рис. 2.16. Обобщения

Реализации обсуждаются в главе 10.

Четвертая – *реализация* (realization) – это семантическая связь между классификаторами, когда один из них специфицирует соглашение, которого второй обязан придерживаться. Вы встретите связи реализации в двух случаях: между интерфейсами и классами или компонентами, которые реализуют эти интерфейсы, а также между вариантами использования и реализующими их кооперациями. Связь реализации в графическом исполнении – гибрид связей обобщения и зависимости (см. рис. 2.17).

Рис. 2.17. Реализации

Эти четыре элемента представляют основные сущности отношений, которые могут быть включены в UML-модели. Есть также различные их вариации: уточнение (refinements), след (trace), включение (include) и расширение (extend).

Пять представлений архитектуры обсуждаются ниже в настоящей главе.

Диаграммы UML. *Диаграмма* – это графическое представление набора элементов, чаще всего изображенного в виде связного графа вершин (сущностей) и путей (связей). Вы рисуете диаграммы для визуализации системы с различных точек зрения, поэтому отдельная диаграмма – это проекция системы. Для всех систем, кроме самых тривиальных, диаграмма представляет собой ограниченный взгляд на элементы, составляющие систему. Один и тот же элемент может появляться либо во всех диаграммах, либо в некоторых (наиболее частый случай), либо вообще ни в одной (очень редкий случай). Теоретически диаграмма может включать в себя любую комбинацию сущностей и связей. На практике, однако, используется лишь небольшое число общих комбинаций, состоящих из пяти наиболее часто применяемых представлений архитектуры программных систем. По этой причине UML включает 13 видов диаграмм:

1. Диаграмма классов.
2. Диаграмма объектов.
3. Диаграмма компонентов.
4. Диаграмма составной структуры.
5. Диаграмма вариантов использования.
6. Диаграмма последовательности.
7. Диаграмма коммуникации.
8. Диаграмма состояний.
9. Диаграмма деятельности.
10. Диаграмма размещения.
11. Диаграмма пакетов.
12. Временная диаграмма.
13. Диаграмма обзора взаимодействий.

Диаграммы классов обсуждаются в главе 8.

Диаграмма классов (class diagram) показывает набор классов, интерфейсов и кооперации, а также их связи. Диаграммы этого вида чаще всего используются для моделирования объектно-ориентированных систем. Предназначены для статического представления системы. Диаграммы классов, включающие активные классы, представляют статическое представление процессов системы. Диаграммы компонентов – это разновидность диаграмм классов.

Диаграммы объектов обсуждаются в главе 14.

Диаграмма объектов (object diagram) показывает набор объектов и их связи. Диаграммы объектов представляют статические копии состояний экземпляров сущностей, описанных в диаграмме классов. Также представляют статическое представление дизайна или статическое представление процессов системы (как и диаграммы классов, но с точки зрения реальных или прототипных ситуаций).

Диаграммы компонентов и внутренние структуры обсуждаются в главе 15.

Диаграмма компонентов (component diagram) демонстрирует инкапсулированные классы и их интерфейсы, порты и внутренние структуры, состоящие из вложенных компонентов и коннекторов. Диаграммы компонентов описывают статическое представление дизайна системы. Они важны при построении больших систем из мелких частей (UML отличает *диаграмму составной структуры* (composite structure diagram), применимую к любому классу, от компонентной диаграммы, но мы рассматриваем их вместе, потому что различие между ними весьма тонкое.)

Диаграммы вариантов использования обсуждаются в главе 18.

Диаграмма вариантов использования (use case diagram) демонстрирует набор вариантов использования и действующих лиц (которые являются специальным видом классов), а также их связи. Диаграммы этого типа описывают статическое представление вариантов использования системы. Особенно важны для организации и моделирования поведения системы.

Диаграммы взаимодействий обсуждаются в главе 19.

И диаграммы последовательностей, и диаграммы коммуникаций являются видами диаграмм взаимодействия. *Диаграмма взаимодействия* (interaction diagram) показывает взаимодействие, состоящее из набора объектов и ролей, включая сообщения, которые могут передаваться между ними. Диаграммы взаимодействия предназначены для динамического представления системы. *Диаграмма последовательности* (sequence diagram) – это разновидность диаграммы взаимодействия, показывающая временную последовательность сообщений. *Диаграмма коммуникаций* (communication diagram) – разновидность диаграммы взаимодействия, показывающая структурную организацию объектов или ролей, отправляющих и принимающих сообщения. И диаграммы последовательности, и диаграммы коммуникации представляют похожие базовые концепции, но с разных точек зрения. Диаграммы последовательности описывают временную последовательность, а коммуникационные диаграммы – структуры данных, через которые проходит поток сообщений.

Диаграммы состояний обсуждаются в главе 25.

Диаграмма состояний (state diagram) показывает автомат (state machine), включающий в себя состояния, переходы, события и деятельности. Диаграммы состояний описывают динамическое представление объекта. Они особенно важны для моделирования поведения интерфейсов, классов или кооперации и подчеркивают событийно-зависимое поведение объекта, что особенно удобно для моделирования реактивных систем.

Диаграммы деятельности обсуждаются в главе 20.

Диаграммы размещения обсуждаются в главе 31.

Диаграммы артефактов обсуждаются в главе 30.

Диаграммы пакетов обсуждаются в главе 12.

Диаграмма деятельности (activity diagram) показывает структуру процесса или других вычислений как пошаговый поток управления и данных. Диаграммы деятельности описываютдинамическое представление системы. Они особенно важны при моделировании функций системы и выделяют поток управления между объектами.

Диаграмма размещения (deployment diagram) показывает конфигурацию узлов-процессоров, а также размещаемые на них компоненты. Диаграммы размещения дают статическое представление размещения архитектуры. Узлы, как правило, содержат один или несколько артефактов.

Диаграмма артефактов (artifact diagram) показывает физический состав компьютерной системы. Артефакты представляют собой файлы, базы данных и подобные им физические наборы битов. Диаграммы данного типа часто применяются в сочетании с диаграммами размещения. Также показывают классы и компоненты, реализованные ими. UML трактует диаграммы артефактов как разновидность диаграмм размещения, но мы рассматриваем их отдельно.

Диаграмма пакетов (package diagram) показывает декомпозицию самой модели на организационные единицы и их зависимости.

Временная диаграмма (timing diagram) – это диаграмма взаимодействий, показывающая реальное время жизни различных объектов или ролей, в противовес простой последовательности сообщений. *Диаграмма обзора взаимодействий* (interaction overview diagram) – это гибрид диаграммы деятельности и диаграммы последовательности. Диаграммы последних двух типов имеют специализированное применение и потому не обсуждаются в данной книге. Подробности читайте в книге «UML» (ее выходные данные приведены во введении, в разделе «Цели»).

Выше мы привели неполный список диаграмм. Инструментальные средства могут использовать UML для представления других типов диаграмм, хотя они не так часто встречаются на практике.

Правила UML

Строительные блоки UML не могут просто быть сброшены в кучу в произвольном порядке. Как и любой язык, UML включает множество правил, указывающих, как должна выглядеть хорошо согласованная модель. *Хорошо согласованной моделью* называется такая, которая семантически самосогласованна и находится в гармонии со всеми другими моделями, связанными с ней.

UML включает синтаксические и семантические правила для:

- имен (как вы можете называть сущности, связи и диаграммы);
- областей действия (контексты, придающие именам специфические значения);

- видимости (как имена могут быть видимы и использованы другими);
- целостности (как сущности правильно и согласованно относятся друг к другу);
- исполнения (что означает запуск или эмуляция динамической модели).

Модели, построенные в процессе разработки программной системы, просматриваются и используются многими заинтересованными лицами для разных целей и в разное время. По этой причине команды разработчиков обычно строят не только хорошо согласованные модели, но и модели, которые:

- имеют умолчания (некоторые элементы скрыты для упрощения представления);
- не полны (некоторые элементы могут быть пропущены);
- не согласованы (целостность модели не гарантирована).

Появление таких не слишком хорошо согласованных моделей неизбежно в процессе разработки, пока не все детали системы прояснились в полной мере. Правила UML не заставляют вас прояснять наиболее важные вопросы анализа, дизайна и реализации, позволяющие построить хорошо согласованные модели, хотя и поощряют к этому.

Общие механизмы UML

Всякое строительство упрощается и ведется более эффективно, если следовать некоторым соглашениям. Дом может быть выстроен в викторианском или французском стиле, если следовать определенным архитектурным образцам. То же верно и в отношении UML. Применение этого языка существенно упрощает последовательное использование механизмов, перечисленных ниже:

- спецификации;
- дополнения;
- принятые разделения;
- механизмы расширения.

Спецификации. UML – нечто большее, чем просто графический язык. За каждой частью его графической нотации стоит *спецификация* (specification), содержащая текстовое представление синтаксиса и семантики определенного строительного блока. Например, пиктограмме класса соответствует спецификация, полностью описывающая набор его атрибутов, операций (включая их полные сигнатуры) и поведение, но визуально пиктограмма может отображать лишь малую часть этой спецификации. Более того, может существовать другое представление этого класса, отражающее совершенно

другие его аспекты, тем не менее соответствующие все той же спецификации. С помощью графической нотации UML вы визуализируете систему, с помощью спецификаций UML описываете ее детали. Таким образом, допускается последовательное построение модели – шаг за шагом, когда сначала рисуются диаграммы, а затем добавляется семантика к спецификациям модели, – или же напрямую, когда в первую очередь создаются спецификации (возможно, при выполнении обратного проектирования существующей системы), а затем рисуются диаграммы, представляющие их проекции.

Спецификации UML создают семантический задний план, который включает в себя все составные части всех моделей системы, согласованные между собой. Таким образом, диаграммы UML – это простые визуальные проекции на этот задний план, при этом каждая из них раскрывает некоторый существенный аспект системы.

Дополнения (adornments). Большинство элементов UML имеют уникальную и прямую графическую нотацию, которая дает визуальное представление наиболее важных аспектов элемента. Например, обозначение класса разработано так, что его легко рисовать, потому что класс – это часто употребляемый элемент при моделировании объектно-ориентированных систем. Нотация класса также показывает его наиболее важные аспекты: имя, атрибуты и операции.

Спецификация класса может содержать и другие детали, такие как видимость атрибутов и операций. Многие из этих деталей могут изображаться в виде графических или текстовых дополнений к базовому прямоугольнику, представляющему класс. Например, рис. 2.18 демонстрирует класс, в обозначение которого включены дополнения, указывающие на то, что этот класс абстрактный, имеет две открытые, одну защищенную и одну закрытую операции.

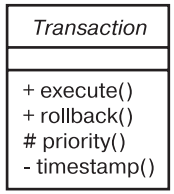


Рис. 2.18. Дополнения

Каждый элемент в нотации UML начинается с базового символа, к которому могут быть добавлены разнообразные специфичные для него дополнения.

Принятые разделения (common divisions). При моделировании объектно-ориентированных систем представление о предметной области может быть разделено несколькими способами.

Объекты обсуждаются в главе 13.

Во-первых, существует разделение на классы и объекты. Класс представляет собой абстракцию, а объект – конкретную ее материализацию. В UML вы можете моделировать как классы, так и объекты (см. рис. 2.19). В графическом представлении объекта UML использует тот же символ, что и для его класса, но с подчеркиванием имени объекта.

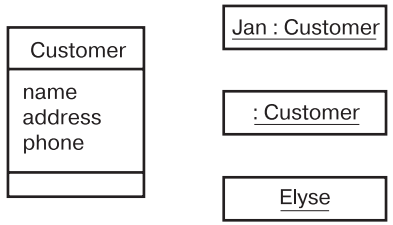


Рис. 2.19. Классы и объекты

На рис. 2.19 показан один класс `Customer` (Покупатель) вместе с тремя объектами: `Jan` (Джен), который помечен, как объект, явно являющийся объектом класса `Customer`, `:Customer` (анонимный объект класса `Customer`) и `Elyse` (Элиза), спецификация которого относит его к классу `Customer`, хотя это не отображено явно.

Почти каждый строительный блок UML характеризуется подобной дихотомией «класс/объект». Например, существуют варианты использования и экземпляры вариантов использования, компоненты и экземпляры компонентов, узлы и экземпляры узлов и т.д.

Интерфейсы обсуждаются в главе 11.

Во-вторых, существует разделение интерфейса и реализации. Интерфейс определяет соглашение, а реализация представляет его конкретное воплощение и обязуется точно следовать полной семантике интерфейса. В UML вы можете моделировать как интерфейсы, так и их реализации (см. рис. 2.20).

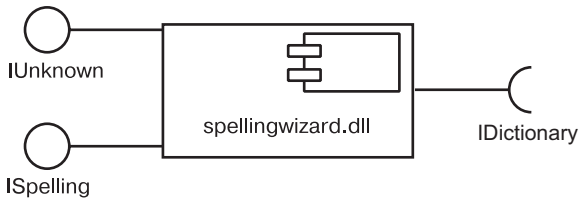


Рис. 2.20. Интерфейсы и реализации

На этом рисунке присутствует один компонент по имени `SpellingWizard.dll` (МастерПроверкиОрфографии.dll), который реализует два интерфейса – `IUnknown` (IНеизвестноеСлово) и `Ispelling` (IНаписание). Он также запрашивает интерфейс `Idictionary` (IСловарь), который должен быть предоставлен другим компонентом.

Почти каждый строительный блок UML обладает той же дихотомией «интерфейс/реализация». Например, варианты использования реализуются кооперациями, а операции – методами.

В-третьих, существует разделение на тип и роль. *Тип* декларирует класс сущности, например объект, атрибут или параметр. Роль описывает значение сущности внутри контекста, такого как класс, компонент или кооперация. Любая сущность, формирующая часть структуры другой сущности (к примеру, атрибут), обладает обеими характеристиками. Она наследует некоторый смысл от типа, которому принадлежит, а другой смысл – от его роли в данном контексте (рис. 2.21).

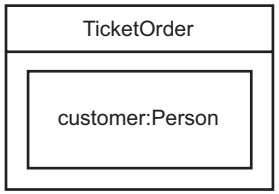


Рис. 2.21. Часть с ролью и типом

Механизмы расширения (extensibility mechanisms). UML – это стандартный язык разработки «чертежей» программного обеспечения, но ни один замкнутый язык не может быть настолько выразительным, чтобы охватить все нюансы всех возможных моделей во всех областях применения в любое время. По этой причине UML является открытым языком, который дает возможность вводить контролируемые расширения. Механизмы расширения UML включают:

- стереотипы;
- помеченные значения;
- ограничения.

Стереотип (stereotype) расширяет словарь UML, позволяя создавать новые виды строительных блоков, которые наследуются от существующих, но при этом специфичны для решения конкретной проблемы. Например, если вы работаете с языком программирования, таким как Java или C++, вам часто приходится моделировать *исключения* (exceptions). В этих языках исключения – это обычные классы, хотя и трактуемые особым образом. Обычно требуется, чтобы исключения можно было возбуждать и перехватывать, и ничего более. Вы можете пометить исключения соответствующим стереотипом, тем самым сделав их подобными базовым строительным блокам. На рис. 2.19 это показано на примере класса `OverFlow`.

Помеченное значение (tagged value) расширяет свойства стереотипа UML, позволяя включать новую информацию в спецификацию

стереотипа. Например, если вы работаете над «коробочным» продуктом и выпускаете много версий, вам часто приходится отслеживать версию и автора определенных важных абстракций. Ни версия, ни автор не являются примитивами UML. Они могут быть добавлены к любому строительному блоку, такому как класс, за счет введения в него новых помеченных значений. Так, например, на рис. 2.19 класс `EventQueue` расширяется явной пометкой его версии и автора.

Ограничения расширяют семантику строительного блока UML, позволяя добавлять новые правила или модифицировать существующие. Предположим, вам понадобилось ограничить класс `EventQueue` так, чтобы все события добавлялись в очередь по порядку. Как видно на рис. 2.22, для этого можно добавить ограничение, явно задающее такое правило для операции add.

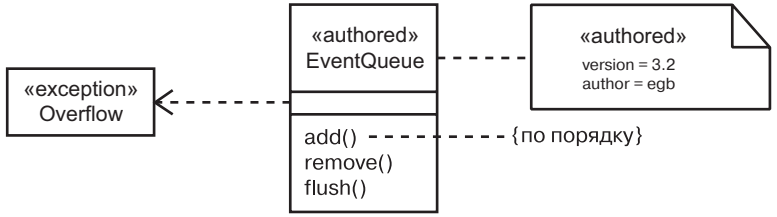


Рис. 2.22. Механизмы расширения

Все эти три механизма расширения языка совместно позволяют модифицировать UML в соответствии с требованиями вашего проекта. Кроме того, они дают возможность адаптировать UML к новым программным технологиям, например к вероятному появлению более мощных языков распределенного программирования. Вы можете добавлять новые строительные блоки, модифицировать спецификацию существующих и даже изменять их семантику. Естественно, при этом нужно соблюдать меру, чтобы подобные расширения не повредили главной цели UML – возможности обмена информацией.

Архитектура

Необходимость представления сложных систем с разных точек зрения обсуждается в главе 1.

Визуализация, специфицирование, конструирование и документирование программных систем требуют их представления с различных точек зрения. Разные заинтересованные лица – конечные пользователи, аналитики, разработчики, системные интеграторы, тестировщики, технические писатели и менеджеры проекта – имеют различное представление о проекте, и каждый из них видит систему по-разному в разные моменты его жизненного цикла. *Системная архитектура* является, пожалуй, наиболее важным продуктом, который может быть использован для управления всеми этими разнообразными точками

зрения и тем самым управляет итеративным и пошаговым процессом разработки системы на протяжении всего ее жизненного цикла.

Архитектура – это набор существенных решений относительно:

- организации программной системы;
- выбора структурных элементов, составляющих систему, и их интерфейсов;
- поведения этих элементов, определенного в их кооперациях;
- объединения этих структурных и поведенческих элементов в более крупные подсистемы;
- архитектурного стиля, определяющего организацию системы: статические и динамические элементы и их интерфейсы, кооперацию и композицию.

Архитектура программного обеспечения касается не только его структуры и поведения, но также пользовательских свойств, функциональности, производительности, гибкости, возможности повторного использования, понятности, экономических и технологических ограничений и компромиссов, а также эстетических вопросов.

Моделирование архитектуры систем обсуждается в главе 32.

Как показано на рис. 2.23, архитектура программной системы может быть наилучшим образом описана с помощью пяти взаимосвязанных представлений. Каждое представление – проекция организации и структуры системы, сосредоточенная на определенном ее аспекте.

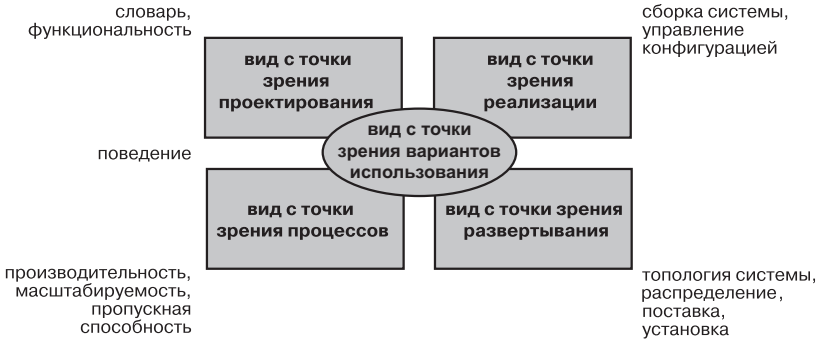


Рис. 2.23. Моделирование системной архитектуры

Представление *вариантов использования* системы охватывает варианты использования, описывающие поведение системы с точки зрения конечных пользователей, аналитиков и тестировщиков. Их взгляд в действительности специфицирует не истинную организацию программной системы, а лишь некие движущие силы, формирующие системную архитектуру. В языке UML статические аспекты этого представления передаются диаграммами вариантов использования, а динамические его аспекты – диаграммами взаимодействий, состояний и деятельности.

Представление системы с точки зрения *дизайна* охватывает классы, интерфейсы и кооперации, формирующие словарь проблемы и ее решение. Это представление в основном поддерживает функциональные требования к системе, то есть сервис, который она должна предоставлять конечным пользователям. Статические аспекты этого представления в UML сосредоточены в диаграммах классов и объектов, а динамические передаются диаграммами взаимодействий, состояний и деятельности. Диаграмма внутренней структуры класса, в частности, также полезна.

Представление *взаимодействия* системы показывает поток управления, проходящий через разные ее части, включая возможные механизмы параллелизма и синхронизации. Это представление касается производительности, масштабируемости и пропускной способности системы. Статические и динамические аспекты этого представления в UML представлены в некоторых видах диаграмм, используемых в представлении дизайна, но сфокусированы на активных классах, управляющих системой, и передаваемых между ними сообщениях.

Представление *реализации* системы охватывает артефакты, используемые для сборки и физической реализации системы. Это представление в первую очередь относится к управлению конфигурацией версий системы, состоящей из независимых (в определенной степени) файлов и компонентов, которые могут быть собраны различными способами для формирования работающей системы. Оно также связано с отображением из логических классов и компонентов в физические артефакты. В UML статические аспекты этого представления отражены в диаграммах артефактов, динамические аспекты – в диаграммах взаимодействия, состояний и деятельности.

Представление *развертывания* системы охватывает узлы, образующие топологию оборудования, на котором работает система. Это представление в основном связано с поставкой, распределением и установкой частей, составляющих физическую систему. Его статические аспекты в UML описываются диаграммами размещения, а динамические – диаграммами взаимодействий, состояний и деятельности.

Каждое из этих пяти представлений может быть достаточным для различных заинтересованных лиц, имеющих отношение к разработке и эксплуатации системы, позволяя им сосредоточиться только на тех аспектах архитектуры, которые непосредственно их касаются. Однако все эти представления также взаимодействуют друг с другом. Узлы из представления размещения содержат компоненты из представления реализации, которые, в свою очередь, представляют собой физическую реализацию классов, интерфейсов, коопераций и активных классов из представлений дизайна и взаимодействия. UML позволяет выразить каждое из пяти представлений.

Жизненный цикл разработки программного обеспечения

Унифицированный процесс разработки программного обеспечения (Rational Unified Process) рассматривается в приложении 2. Более подробное его описание приводится в «Унифицированном процессе разработки программного обеспечения» и в «The Rational Unified Process».

Язык UML в основном независим от организации процесса разработки, то есть не привязан к какому-либо конкретному циклу производства программного продукта, но для того, чтобы в максимальной степени воспользоваться его преимуществами, вам стоит рассмотреть процесс, который должен быть:

- управляемым вариантами использования;
- сконцентрированным на архитектуре;
- итеративным и пошаговым.

Процесс, *управляемый вариантами использования* (use case driven) – означает, что варианты использования используются в качестве первичных рабочих продуктов, на основании которых определяется желательное поведение системы, проверяется и подтверждается правильность выбранной системной архитектуры, производится тестирование и ведется общение заинтересованных лиц, имеющих отношение к проекту.

Процесс, *сконцентрированный на архитектуре* (architecture-centric) – означает, что архитектура системы является первичным рабочим продуктом для процесса концептуализации, конструирования, управления и развития системы во время ее разработки.

Итеративный (iterative) процесс – тот, который включает управление потоком исполняемых версий системы. *Пошаговый* (incremental) процесс подразумевает непрерывную интеграцию системной архитектуры в целях выпуска версий, каждая последующая из которых усовершенствована по сравнению с предыдущей. Процесс, являющийся итеративным и пошаговым, называется *процессом с управляемым риском* (risk-driven), поскольку при выпуске каждой новой версии (релиза) серьезное внимание уделяется выявлению факторов, представляющих наибольший риск для проекта в целом, и сведения их к минимуму.

Такой управляемый вариантами использования, сконцентрированный на архитектуре, итеративный и пошаговый процесс может быть разбит на фазы. *Фаза* – это отрезок времени между двумя важными *контрольными точками* (milestones) процесса, в которых достигаются четко определенные цели, завершено создание рабочих продуктов и принимается решение о переходе к следующей фазе. Как показано на рис. 2.24, существуют четыре фазы жизненного цикла разработки: начальная фаза, разработка, конструирование и внедрение. На диаграмме поток работ изображен в контексте этих фаз, в результате чего демонстрируется различный уровень концентрации на выполнении его составных частей.

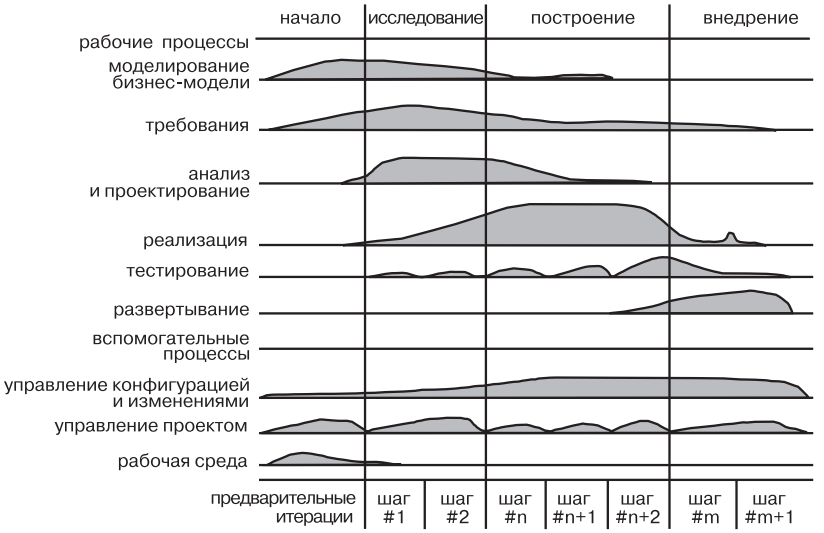


Рис. 2.24. Жизненный цикл разработки программной системы

Начальная фаза (inception) – первая фаза процесса, на протяжении которой начальная идея получает достаточное обоснование (по крайней мере, внутреннее) для обеспечения перехода к фазе разработки.

Разработка (elaboration) – вторая фаза процесса, когда определяются требования к продукту и архитектуре. Системные требования могут включать в себя широкий диапазон формулировок – от общих предложений до детальных критериев оценки, каждый из которых посвящен описанию определенного функционального или нефункционального поведения системы и закладывает основы для организации тестирования.

Конструирование (construction) – третья фаза процесса, когда исполняемый архитектурный прототип приобретает форму, в которой он может быть представлен пользователям. На этом этапе требования к системе, особенно критерии оценки, постоянно пересматриваются в соответствии с изменяющимися потребностями. Также выделяются необходимые ресурсы для уменьшения рисков.

Внедрение (transition) – четвертая фаза процесса, в ходе которой программное обеспечение передается пользователям. Но процесс разработки редко завершается на этом, потому что даже на данной фазе система непрерывно совершенствуется: устраняются ошибки и добавляются не вошедшие в ранние версии новые функциональные возможности.

Один элемент, который отличает процесс разработки в целом и присутствует во всех четырех фазах, – это итерация. *Итерация* – четко определенная последовательность действий с ясно сформулированным

планом и критериями оценки, приводящая к появлению новой версии системы, которая может быть выполнена, протестирована и оценена. Исполняемая система не обязана быть реализована внешне. Поскольку итерация порождает исполняемый продукт, достигнутый прогресс и уровень риска могут быть оценены заново после каждой такой итерации. Это значит, что жизненный цикл разработки программного обеспечения можно характеризовать как непрерывный поток исполняемых версий, реализующих архитектуру системы с промежуточной коррекцией для снижения потенциального риска. Это подчеркивает значение архитектуры как важнейшего элемента программной системы и фокусирует UML на моделировании различных ее представлений.

Глава 3. Здравствуй, мир!

В этой главе:

- Классы и артефакты
- Статические и динамические модели
- Взаимодействие моделей
- Расширения UML

Брайан Керниган (Brian Kernighan) и Деннис Ричи (Dennis Ritchie), авторы языка программирования C, указывали, что «единственный способ изучить новый язык программирования – это писать на нем программы». То же верно и в отношении UML. Единственный способ изучить этот язык – писать модели на нем.

Первая программа, которую многие разработчики пишут, приступая к освоению нового языка программирования, – простейшая: она делает немногим больше, чем просто печатает строку «Здравствуй, мир!» (Hello, World!) Вполне разумное начало, потому что, справившись с таким тривиальным приложением, вы наверняка захотите продолжать учиться. Кроме того, эта программа позволяет раскрыть всю необходимую инфраструктуру, необходимую для ее сборки и запуска.

Итак, начнем изучение UML с моделирования «Здравствуй, мир!» Это почти самое простое применение языка, которое только можно себе представить. Однако простота обманчива: в основе приложения, которое мы сейчас создадим, лежат некоторые интересные механизмы, обеспечивающие его работу.

Ключевые абстракции

Апплет на языке Java, выводящий строку «Здравствуй, мир!» в Web-браузере, достаточно прост:

```
import java.awt.Graphics;
class HelloWorld extends java.applet.Applet {
    public void paint (Graphics g) {
      g.drawString("Здравствуй, мир!", 10, 10);
    }
}
```

Первая строка кода

```
import java.awt.Graphics;
```

обеспечивает доступ к классу Graphics (Графика) для последующего кода. Префикс java.awt. специфицирует пакет Java, в котором находится класс Graphics.

Вторая строка кода

```
class HelloWorld extends java.applet.Applet {
```

представляет новый класс по имени HelloWorld и указывает, что он является потомком класса Applet, находящегося в пакете java.applet.

Следующие три строки кода:

```
public void paint (Graphics g) {
g.drawString("Здравствуй, мир!", 10, 10);
}
```

объявляют операцию по имени paint, реализация которой вызывает другую операцию– drawString (выводСтроки), отвечающую за вывод строки «Здравствуй, мир!» по заданным координатам. В обычной объектно-ориентированной манере drawString является операцией объекта, переданного в параметре g, тип которого – класс Graphics.

Классы обсуждаются в главах 4 и 9.

Моделирование этого приложения на UML достаточно просто. Как показывает рис. 3.1, вы можете представить класс HelloWorld графически в виде прямоугольной пиктограммы. Его операция paint (рисование) показана здесь же, причем ее формальные параметры опущены, а реализация указана в присоединенном примечании.

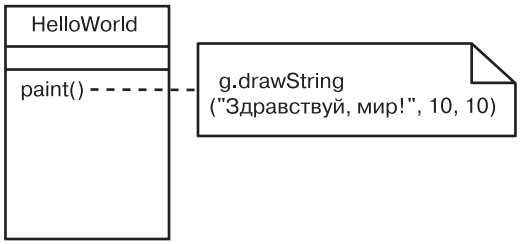


Рис. 3.1. Ключевые абстракции HelloWorld

На заметку. UML не является визуальным языком программирования, хотя, как видно по рисунку, он позволяет (но не требует) обеспечить тесную связь с разными языками программирования, такими как Java. UML позволяет трансформировать модели в код и обратно – из кода в модель. Некоторые детали (например, математические выражения) при этом лучше выразить на текстовом языке программирования, между тем как другие (например, иерархию классов) – визуализировать графически на UML.

Эта диаграмма классов передает основную суть приложения «Здравствуй, мир!», но в то же время оставляет без внимания множество вещей. Как специфицирует приведенный выше код, в приложении участвуют два других класса – Applet (Апплет) и Graphics (Графика), причем они используются разными способами. Класс Applet используется в качестве родительского для HelloWorld, а класс Graphics используется в сигнатуре и реализации его операции print. Вы можете представить эти классы и их различные связи с классом HelloWorld на диаграмме классов, как показано на рис. 3.2.

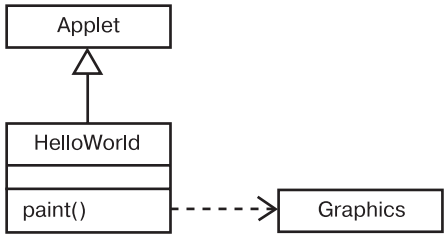


Рис. 3.2. Ближайшие соседи, окружающие HelloWorld

Связи обсуждаются в главах 5 и 10.

Классы Applet и Graphics изображены в виде прямоугольных пиктограмм. Никакие операции для них не показаны, поэтому эти пиктограммы пусты. Линия с пустой стрелкой от HelloWorld к Applet представляет обобщение, которое в данном случае означает, что HelloWorld является дочерним по отношению к Applet. Пунктирная линия, напраленная от HelloWorld к Graphics, представляет связь зависимости, означающую, что HelloWorld использует Graphics.

И это еще не вся структура, лежащая в основе HelloWorld. Если вы изучите библиотеки Java, содержащие Applet и Graphics, то обнаружите, что оба этих класса являются частями более крупной иерархии. Просматривая только те классы, которые расширяет и реализует Applet, вы сможете нарисовать другую диаграмму классов, показанную на рис. 3.3.

На заметку. На рис. 3.3 представлен пример диаграммы, сгенерированной в результате обратного проектирования существующей системы. Обратное проектирование – это создание модели на основании кода.

По рисунку становится видно, что HelloWorld – только «листик» в большой иерархии классов. HelloWorld – дочерний класс Applet; Applet – дочерний класс Panel (Панель); Panel – дочерний по отношению к Container (Контейнер); Container – дочерний класс Component (Компонент), а Component – дочерний по отношению к Object (Объект), который является родительским классом для всех классов Java. Таким образом, эта модель соответствует организации библиотеки Java – каждый дочерний класс расширяет (наследует) свойства некоторого родителя.

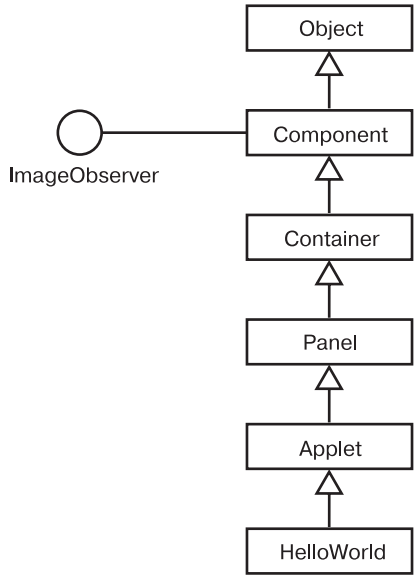


Рис. 3.3. Иерархия наследования `HelloWorld`

Интерфейсы обсуждаются в главе 11.

Связь между `ImageObserver` (ОбозревательИзображений) и `Component` несколько отличается от предыдущих связей обобщения, и диаграмма классов отражает это отличие. В библиотеке Java `ImageObserver` – это интерфейс; отсюда следует, в частности, что он сам по себе не имеет реализации, а вместо этого требует, чтобы другие классы реализовывали его. Вы можете показать, что класс `Component` реализует интерфейс `ImageObserver`, с помощью сплошной линии, проведенной от прямоугольника (`Component`) к кружку, который символизирует интерфейс (`ImageObserver`).

Как показано на рис. 3.3, `HelloWorld` непосредственно взаимодействует только с двумя классами (`Applet` и `Graphics`), которые являются лишь малой частью большой библиотеки предопределенных классов Java. Чтобы управлять этой большой коллекцией, Java организует свои интерфейсы и классы во множестве пакетов. Корневой пакет среды Java носит имя `java` (ну кто бы мог подумать!) В него вложено несколько пакетов, содержащих, в свою очередь, другие пакеты, интерфейсы и классы. `Object` «живет» в пакете `lang`, поэтому полный путь к нему выглядит так: `java.lang.Object`. Аналогичным образом `Panel`, `Container` и `Component` находятся в пакете `awt`, класс `Applet` – в пакете `applet`. Интерфейс `ImageObserver` находится в пакете `image`, который, в свою очередь, размещен в `awt`, поэтому полный путь к нему – `java.awt.image.ImageObserver`.

Пакеты обсуждаются в главе 12.

Вы можете представить такое пакетирование на диаграмме классов, показанной на рис. 3.4. Как следует из рисунка, пакеты представлены

в UML в виде папок с закладками. Пакеты могут быть вложенными, и пунктирные линии со стрелками выражают зависимости между ними. Например, `HelloWorld` зависит от пакета `java.applet`, а `java.applet` – от `java.awt`.

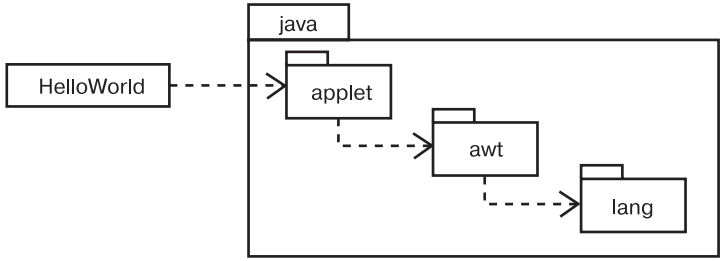


Рис. 3.4. Пакетирование `HelloWorld`

Механизмы

Образцы и каркасы обсуждаются в главе 29.

Наиболее трудная задача, возникающая при освоении такой богатой библиотеки, как библиотека языка Java, – понять, как ее части работают вместе. Например, как вызывается операция `paint` класса `HelloWorld`? Какие операции следует использовать, чтобы изменить поведение этого апплета – скажем, напечатать строку другого цвета? Для ответа на эти и другие вопросы вы должны иметь концептуальную модель, показывающую, как классы работают совместно в динамике.

Процессы и потоки рассматриваются в главе 23.

Изучение библиотеки Java показывает, что операция `paint` класса `HelloWorld` наследуется от `Component`. Но остается открытым вопрос, как она вызывается. Ответ в том, что `paint` вызывается как часть потока, в котором исполняется апплет (см. рис. 3.5).

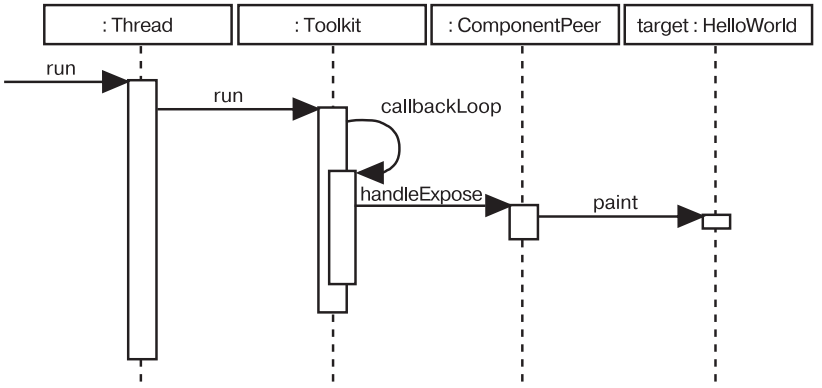


Рис. 3.5. Изображение механизма

Экземпляры обсуждаются в главе 13.

На рисунке показана кооперация нескольких объектов, включая один экземпляр класса `HelloWorld`. Прочие изображенные здесь объекты являются частью среды Java, поэтому в основном находятся на заднем плане создаваемых вами апплетов. Все это демонстрирует кооперацию между объектами, которая может применяться многократно. Каждый столбец показывает роль в кооперации, то есть часть, которая может быть исполнена разными объектами при каждом запуске. В UML роли изображаются так же, как классы, с тем отличием, что для них указываются и тип, и имя роли. Средние две роли на данной диаграмме являются анонимными, потому что их типа достаточно для того, чтобы идентифицировать их в пределах кооперации (но двоеточие и отсутствие подчеркивания указывают на то, что это роли). Начальный `Thread` называется `root`, а роль `HelloWorld` именована `target` и известна роли `ComponentPeer`.

Диаграммы последовательности рассматриваются в главе 19.

Вы можете моделировать порядок событий, используя диаграмму последовательности, изображенную на рис. 3.5. Здесь последовательность начинается с запуска объекта `Thread`, который, в свою очередь, вызывает операцию `run` объекта `Toolkit`. Объект `Toolkit` затем вызывает одну из его собственных операций (`callbackLoop`), которая затем вызывает операцию `handleExpose` объекта `ComponentPeer`. Объект `ComponentPeer` предполагает, что его цель является экземпляром `Component`, но в данном случае это объект дочернего класса `Component`, а именно `HelloWorld`, поэтому полиморфно вызывается операция `paint` класса `HelloWorld`.

Артефакты

Программа «Здравствуй, мир!» реализована в виде апплета, поэтому она никогда не запускается самостоятельно, а только в составе некоторой Web-страницы. Апплет стартует, когда содержащая его страница открывается механизмом браузера, запускающим объект `Thread` этого апплета. Однако не существует класса `HelloWorld`, который был бы непосредственной частью Web-страницы. Имеется лишь бинарная форма этого класса, созданная компилятором Java, который трансформировал исходный код, представляющий этот класс в исполняемый артефакт. Таким образом, формируется совершенно другой взгляд на систему. В то время как все предшествовавшие диаграммы предлагали логическое представление апплета, то, что мы видим теперь, – представление его физических артефактов.

Артефакты обсуждаются в главе 26.

Вы можете смоделировать это физическое представление с помощью диаграммы артефактов, как показано на рис. 3.6.

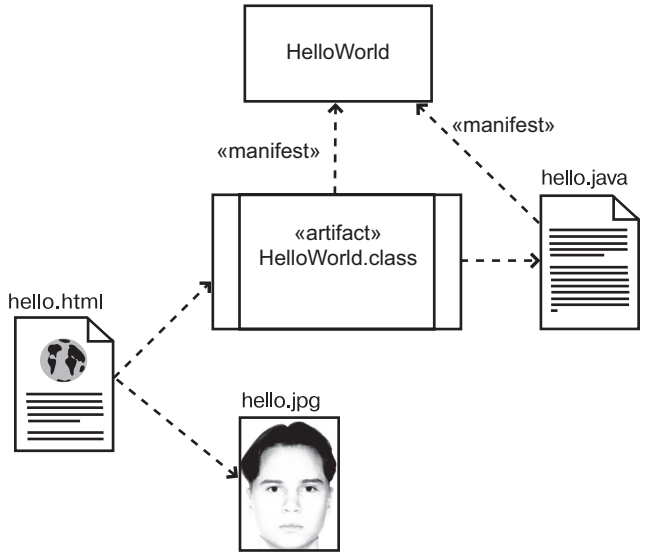


Рис. 3.6. Артефакты `HelloWorld`

Логический класс `HelloWorld` показан в верхнем прямоугольнике. Все другие пиктограммы на рисунке символизируют артефакты UML в представлении реализации системы. *Артефакт* – это физическая сущность, такая как файл. Артефакт по имени `hello.java` представляет исходный код логического класса `HelloWorld`; этим файлом могут манипулировать среда разработки и инструменты управления конфигурацией. Исходный код может быть трансформирован в бинарный апплет `hello.class` с помощью компилятора Java, что делает его пригодным для запуска в среде виртуальной машины Java. Как исходный текст, так и бинарный апплет физически реализуют логический класс. Это показано пунктирными стрелками, помеченными ключевым словом `«manifest»`.

Механизмы расширения UML обсуждаются в главе 6.

Пиктограмма артефакта представляет собой прямоугольник с ключевым словом `«artifact»`, расположенным над именем. Бинарный апплет `HelloWorld.class` – вариация этого базового символа, но с утолщенными линиями, указывающими на то, что это исполняемый артефакт (подобно активному классу). Пиктограмма артефакта `hello.java` заменена пользовательской иконкой, представляющей текстовый файл. Пиктограмма Web-страницы `hello.html` оформлена аналогичным образом, с расширением нотации UML. Как следует из рисунка, эта Web-страница содержит другой артефакт, `hello.jpg`, который тоже представлен пользовательской иконкой – теперь уже графическим файлом. Так как последние три артефакта представлены определяемыми пользователем графическими символами, их

имена помещены вне пиктограмм. Зависимости между артефактами изображены пунктирными стрелками.

На заметку. Связи между классом (`HelloWorld`), его исходным кодом (`hello.java`) и объектным кодом (`HelloWorld.class`) редко моделируются явным образом, хотя иногда бывает удобно сделать это, чтобы визуализировать физическую конфигурацию системы. С другой стороны, общепринятой практикой является визуальное представление системы, основанной на Web, как в приведенном примере – с использованием диаграмм артефактов для моделирования страниц и других исполняемых артефактов.

Часть II

Основы структурного моделирования

Глава 4. Классы

В этой главе:

- Классы, атрибуты, операции и обязанности
- Моделирование словаря системы
- Моделирование распределения обязанностей в системе
- Моделирование непрограммных сущностей
- Моделирование примитивных типов
- Создание качественных абстракций

Классы – наиболее важные строительные блоки любой объектно-ориентированной системы. Класс – это описание множества объектов с одинаковыми атрибутами, операциями, связями и семантикой. Класс реализует один или несколько интерфейсов.

Дополнительные свойства классов обсуждаются в главе 9.

Вы используете классы, чтобы зафиксировать словарь разрабатываемой системы. Эти классы могут включать абстракции, являющиеся частью проблемной области, а также другие классы, составляющие реализацию. Можно применять классы для представления программных сущностей, аппаратных сущностей и даже тех сущностей, которые являются полностью концептуальными.

Хорошо структурированные классы имеют четко очерченные границы и формируют сбалансированное распределение обязанностей в системе.

Введение

Моделирование системы включает в себя идентификацию сущностей, которые важны для вашего конкретного представления. Эти сущности формируют словарь системы, которую вы моделируете. Например, при построении дома вам как будущему владельцу небезразлично, какими будут стены, двери, окна, кабинеты, системы освещения и пр. Каждая из этих сущностей отличается от другой и обладает набором определенных свойств. Стены имеют некую высоту, ширину и являются сплошными. Двери характеризуются теми же признаками, но их к тому же отличает специфическое *поведение*: они открываются в одну сторону. Окна отчасти схожи

с дверьми, поскольку тоже образуют проем в стене, но их функциональность неодинакова. Обычно (хотя и не всегда) окна предназначены для того, чтобы вы смотрели через стекло на улицу.

Отдельные стены, двери и окна редко существуют сами по себе, поэтому вы должны также рассмотреть, как конкретные экземпляры этих сущностей сочетаются друг с другом. Сущности, которые вы идентифицируете, и связи между ними будут зависеть от того, как вы собираетесь использовать различные комнаты вашего дома, как планируете организовать движение из одной комнаты в другую, а также от общего дизайнерского решения.

Рабочие, которые строят дом, сосредоточены на других вещах. Например, водопроводчик, который помогает разрабатывать поэтажный план, заинтересован в расположении водостоков, сифонов и вентиляционных отверстий. Вы как домовладелец не обязательно принимаете в расчет такие вещи (за исключением тех случаев, когда оказываются затронуты важные для вас сущности, – например, когда водосток может быть устроен в полу или вентиляция выходит через крышу).

Объекты обсуждаются в главе 13.

В UML все эти сущности моделируются классами. *Класс* – это абстракция сущности, являющейся частью вашего словаря. Класс – не индивидуальный объект, а представление целого множества объектов: скажем, вы можете концептуально представлять «стену» как класс объектов с рядом общих свойств (высота, длина, толщина; является ли стена несущей и т.д.). Но можно иметь в виду и вполне конкретную стену, например находящуюся в юго-западном углу вашей гостиной.

Что же касается программного обеспечения, многие языки программирования непосредственно поддерживают концепцию класса. Это очень удобно, потому что создаваемые вами абстракции часто могут быть напрямую выражены на языке программирования, даже если речь идет об абстракциях непрограммных сущностей, таких как заказчик, торговля или переговоры.

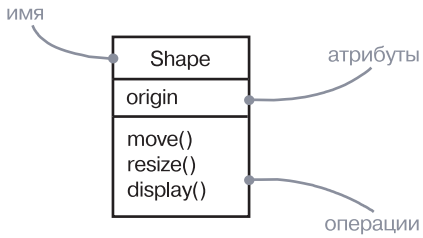


Рис. 4.1. Классы

UML предлагает графическое представление класса (см. рис. 4.1). Эта нотация позволяет вам визуализировать абстракцию вне какого-либо определенного языка программирования, притом подчеркнув ее наиболее важные части: имя, атрибуты и операции.

Базовые понятия

Класс (class) – это описание множества объектов с одинаковыми атрибутами, операциями, связями и семантикой. Изображается в виде прямоугольника.

Имена

Имя класса должно быть уникально в пределах включающего его пакета (см. главу 12).

Каждый класс должен обладать именем, отличающим его от других классов. *Имя* (name) – это текстовая строка. Отдельное имя, взятое само по себе, называется *простым*, а имя класса с префиксом – именем пакета, в котором «живет» этот класс, – *квалифицированным*. Класс можно изобразить только с указанием его имени, как показано на рис. 4.2.

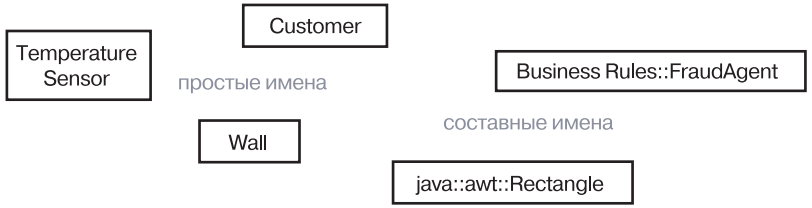


Рис. 4.2. Простые и квалифицированные имена

На заметку. Имя класса может состоять из любого числа букв, цифр и знаков препинания (за исключением таких символов, как двоеточие или точка, которая применяется для отделения имени класса от имени содержащего его пакета) и записываться в несколько строк. На практике обычно используются краткие имена классов – существительные, взятые из словаря моделируемой системы. Каждое слово в имени класса традиционно пишут с заглавной буквы, например `Customer` (Покупатель) или `TemperatureSensor` (ДатчикТемпературы).

Атрибуты

Атрибуты имеют отношение к семантике агрегации (см. главу 10).

Атрибут (attribute) – это именованное свойство класса, описывающее диапазон значений, которые может принимать экземпляр атрибута. Класс может иметь любое число атрибутов или не иметь ни одного. Атрибут представляет некоторое свойство моделируемой сущности, которым обладают все объекты данного класса: например, у каждой стены есть высота, ширина и толщина. Аналогичным образом вы можете описать своих заказчиков: у каждого из них есть имя, адрес, номер телефона, дата рождения. Таким образом,

атрибут – это абстракция вида данных или состояния, которое может принимать объект того или иного класса. В каждый определенный момент объект класса характеризуется конкретными значениями каждого из атрибутов. В графическом представлении атрибуты перечислены в разделе, приведенном непосредственно под именем класса, причем с обязательным указанием их имен (см. рис. 4.3).

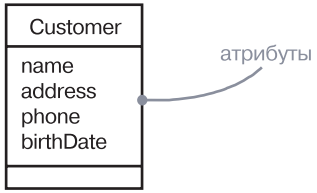


Рис. 4.3. Атрибуты

На заметку. Имя атрибута, как и имя класса, может представлять собой текст. На практике для именования атрибута используются одно или несколько коротких существительных, выражающих некое свойство класса, к которому относится атрибут. Обычно каждое слово в имени атрибута пишется с заглавной буквы, за исключением первого, например `name` (имя) или `birthDate` (датаРождения).

Вы можете задать другие свойства атрибута, например снабдить его пометкой `readonly` *(только для чтения) или объявить общим для всех объектов класса – см. главу 9.*

Можно уточнить спецификацию атрибута, указав его класс и начальное значение по умолчанию, как показано на рис. 4.4.

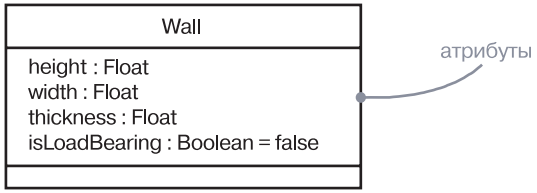


Рис. 4.4. Атрибуты и их класс

Операции

Операция (operation) – это реализация услуги, которая может быть запрошена у любого объекта данного класса, чтобы вызвать определенное его поведение. Другими словами, операция – это абстракция чего-либо, что вы можете сделать с конкретным объектом и вообще со всеми объектами данного класса. Класс может иметь любое число операций либо не иметь ни одной. Например, в оконной библиотеке наподобие той, что содержится в пакете Java awt, все

Реализация операции уточняется при помощи примечаний (см. главу 6) либо диаграммы деятельности (см. главу 20).

объекты класса Rectangle можно перемещать, изменять их размер или запрашивать их свойства. Часто (хотя и не всегда) вызов операции объекта изменяет его данные или состояние. Графически операции представлены в разделе списка, приведенного под атрибутами класса. Допускается указание только имен операций (см. рис. 4.5).

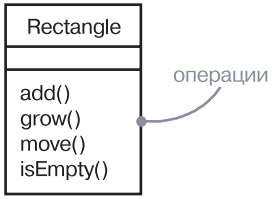


Рис. 4.5. Операции

На заметку. Имя операции, как и имя класса, может представлять собой текст. На практике для именования операции используются короткие глагольные конструкции, описывающие некое поведение класса, которому принадлежит операция. Обычно каждое слово в имени операции пишется с заглавной буквы, за исключением первого, например `move` (переместить) или `isEmpty` (пуст).

Вы можете уточнить другие свойства операции, например пометив ее как полиморфную или константную либо указав ее видимость (см. главу 9).

Можно специфицировать операцию, устанавливая ее сигнатуру, включающую имя, тип и значение по умолчанию всех параметров, а применительно к функциям – тип возвращаемого значения (рис. 4.6).

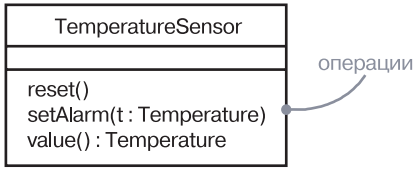


Рис. 4.6. Операции и их сигнатур

Организация атрибутов и операций

Изображая класс, вы не обязаны показывать сразу все его атрибуты и операции. Фактически в большинстве случаев вы не в состоянии этого сделать (таких элементов может быть слишком много, чтобы все они поместились на одном рисунке), да и не должны (для конкретного представления, как правило, существенна только часть атрибутов и операций класса). В силу этих причин допускается упрощенное представление класса, то есть для графического

представления выбираются только некоторые из его атрибутов и операций – или же они вообще не изображаются. Если помимо указанных существуют другие атрибуты и операции, вы даете это понять, завершая каждый список многоточием. Можно и вовсе не обозначать соответствующий раздел на пиктограмме класса – тогда не будет видно, имеет ли класс атрибуты или операции, сколько их и т.п.

Стереотипы обсуждаются в главе 6.

Чтобы лучше организовать длинные списки атрибутов и операций, желательно снабдить префиксом (именем стереотипа) каждую категорию в них, как показано на рис. 4.7.

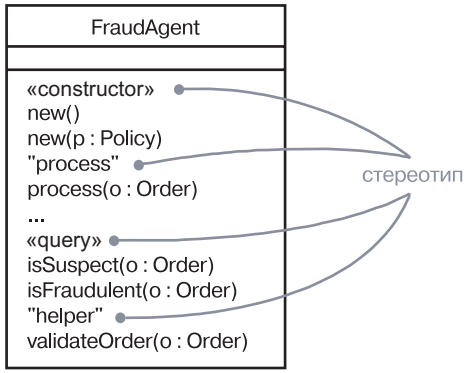


Рис. 4.7. Стереотипы в свойствах класса

Обязанности

Обязанности – это пример определенных стереотипов (см. главу 6).

Обязанность (responsibility) – это соглашение или обязательства класса. Когда вы создаете класс, выдвигается предположение, что все его объекты характеризуются одинаковым состоянием и одинаковым поведением. На более абстрактном уровне соответствующие атрибуты и операции являются просто средствами, благодаря которым класс выполняет свои обязанности. Класс Wall (Стена) отвечает за информацию о высоте, ширине, толщине; класс FraudAgent (АгентПоПредотвращениюМошенничеств), который можно встретить в приложении, обрабатывающем кредитные карты, – за обработку запросов и определение их обоснованности, подозрительности или незаконности; класс TemperatureSensor (ДатчикТемпературы) – за измерение температуры и подачу сигнала тревоги в случае превышения допустимого предела.

Моделирование семантики класса обсуждается в главе 9.

Когда вы моделируете классы, для начала очень неплохо сформулировать обязанности сущностей из вашего словаря. Методика применения CRC-карт[2] и анализа на основе вариантов использования

[2] Карты CRC (Class–Responsibility–Collaboration – «класс, обязанности, взаимодействие») – один из широко применяемых в объектно-ориентированном анализе методов распределения обязанностей между классами. – *Прим. ред.*

может здесь оказаться особенно полезной. У класса может быть сколько угодно обязанностей, хотя на практике каждый хорошо структурированный класс имеет как минимум одну обязанность и как максимум – небольшой их набор. По мере детализации ваших моделей вы будете отображать эти обязанности на множество атрибутов и операций, которые наилучшим образом им соответствуют.

Обязанности класса можно также описать в примечании (см. главу 6).

Графически обязанности могут быть представлены в специально отведенном для них разделе, в нижней части пиктограммы класса (см. рис. 4.8).

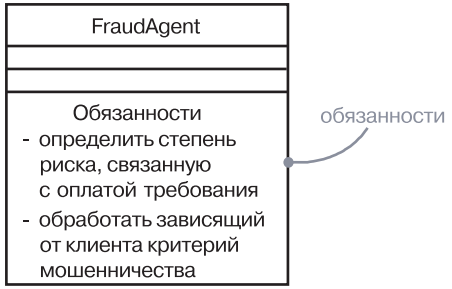


Рис. 4.8. Обязанности

На заметку. Обязанности выражаются текстом в свободном формате. Отдельная обязанность представлена фразой, предложением или (как максимум) коротким абзацем.

Прочие характеристики

Более сложные конструкции класса обсуждаются в главе 9.

Атрибуты, операции и обязанности – это наиболее часто используемые средства, которые понадобятся вам при создании абстракций. Фактически для большинства моделей, которые вы строите, базовая форма этих трех компонентов – все, что нужно для выражения самых важных частей семантики ваших классов. Однако иногда у вас будет возникать потребность в визуализации или спецификации некоторых других характеристик (таких как видимость отдельных атрибутов операций); свойств операций, зависимых от языка (таких как полиморфизм или константность); либо даже исключений, которые объект класса может генерировать или обрабатывать. Эти и многие другие средства могут быть выражены средствами UML, но они рассматриваются как дополнительные понятия.

Интерфейсы обсуждаются в главе 11.

Приступив к построению моделей, вы очень скоро обнаруживаете, что почти каждая абстракция, которую вы создаете, представляет собой класс определенного вида. Иногда реализацию класса необходимо отделять от его спецификации. В UML это может быть выражено посредством интерфейсов.

Внутренняя структура классов обсуждается в главе 15.

При проектировании реализации класса вам потребуется моделировать ее внутреннюю структуру как набор взаимосвязанных частей. Чтобы расставить все точки над i, вы можете разделить внутреннюю структуру класса верхнего уровня на несколько слоев.

Активные классы, компоненты и узлы обсуждаются в главах 23, 25 и 27, артефакты – в главе 26.

Когда вы перейдете к проектированию более сложных моделей, то обнаружите, что вам все чаще приходится иметь дело с некоторыми новыми сущностями: классами, представляющими параллельные процессы и потоки, либо классификаторами, описывающими физические сущности, такие как апплеты, компоненты JavaBeans, файлы, Web-страницы и аппаратное обеспечение. Поскольку сущности подобного рода встречаются очень часто и представляют собой важные архитектурные абстракции, для их моделирования в UML предусмотрены активные классы (описывающие процессы и потоки), классификаторы, в частности артефакты (описывающие физические компоненты программного обеспечения), и узлы (описывающие аппаратные устройства).

Диаграммы классов обсуждаются в главе 8.

Наконец, классы редко существуют сами по себе. Когда вы строите модели, то, как правило, фокусируете внимание на *группах классов*, взаимодействующих друг с другом. В UML такие сообщества классов формируют кооперации и обычно визуализируются в диаграммах классов.

Типичные приемы моделирования

Моделирование словаря системы

Варианты использования обсуждаются в главе 17.

Чаще всего вы будете использовать классы для моделирования абстракций, происходящих от проблем, которые вы пытаетесь решить, либо от технологий, используемых для решений этих проблем. Каждая из таких абстракций является частью *словаря* вашей системы, то есть вместе взятые, они представляют совокупность понятий, важных для пользователей и разработчиков.

Для пользователей идентифицировать большинство абстракций не так сложно, потому что последние, как правило, происходят от тех элементов и понятий, которые уже используются для описания систем. Методика применения CRC-карт и анализа на основе вариантов использования – отличный способ помочь найти эти абстракции. Что же касается разработчиков, для них подобные абстракции обычно являются всего лишь технологическими сущностями, представляющими части решения.

Чтобы смоделировать словарь системы, необходимо:

- Идентифицировать те сущности, которыми оперируют пользователи или разработчики для описания проблемы или ее

решения. Использовать CRC-карты и анализ на основе вариантов использования для выявления этих абстракций.

- ❑ Для каждой абстракции идентифицировать набор ее обязанностей. Убедиться, что каждый класс четко определен и обязанности сбалансированно распределены между классами.
- ❑ Представить атрибуты и операции, необходимые для выполнения обязанностей каждого класса.

Рис. 4.9 показывает набор классов, предназначенный для реализации системы розничной торговли и содержащий классы `Customer` (Покупатель), `Order` (Заказ) и `Product` (Продукт). Также рисунок включает несколько других абстракций, связанных с основными, – они взяты из словаря проблемной области. Это `Shipment` (Поставка) – используется для отслеживания заказов; `Invoice` (Счет-фактура) – для оплаты заказов; `Warehouse` (Склад) – место хранения товаров перед отправкой. Есть также одна абстракция, имеющая отношение к решению: `Transaction` (Транзакция). Она применяется к заказам и поставкам.

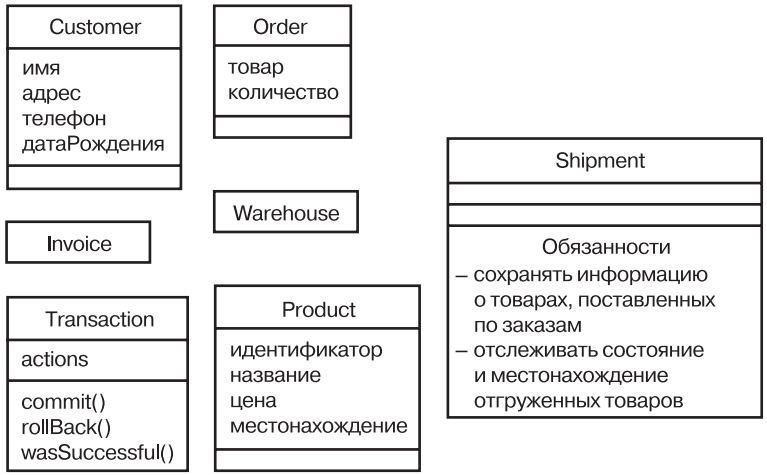


Рис. 4.9. Моделирование словаря системы

Пакеты обсуждаются в главе 12.

По мере расширения ваших моделей вы обнаружите, что многие классы имеют тенденцию собираться в группы (кластеры), концептуально и семантически связанные. В UML для моделирования таких кластеров можно использовать пакеты.

Моделирование поведения обсуждается в главах 4 и 5.

Редко когда ваши модели будут абсолютно статическими. Скорее всего, абстракции из словаря вашей системы в большинстве своем будут динамически взаимодействовать друг с другом. UML предусматривает множество способов моделирования динамического поведения.

Моделирование распределения обязанностей в системе

Если вам придется моделировать немало классов, вы захотите убедиться, что ваши абстракции обеспечивают сбалансированный набор обязанностей. Иными словами, нельзя допустить, чтобы какой-то из классов оказался слишком большим или слишком маленьким. Каждый класс должен хорошо выполнять одну задачу. Если вы абстрагируете классы, которые окажутся слишком большими, то обнаружите, что ваша модель трудно поддается изменениям и не слишком удобна для повторного использования. Если же абстрагировать слишком маленькие классы, у вас будет слишком много абстракций, которые трудно понять и которыми трудно управлять. UML поможет визуализировать и специфицировать баланс обязанностей.

Чтобы смоделировать распределение обязанностей в системе, необходимо:

- ❑ Идентифицировать множество классов, работающих совместно для достижения некоторого поведения.
- ❑ Идентифицировать набор обязанностей каждого из этих классов.
- ❑ Рассмотреть множество классов как одно целое; расчленить классы, на которые приходится слишком большая доля обязанностей, на меньшие абстракции; объединить мелкие классы с тривиальными обязанностями в более крупные; перераспределить обязанности так, чтобы для каждой абстракции был отведен их оптимальный набор.

Кооперации обсуждаются в главе 28.

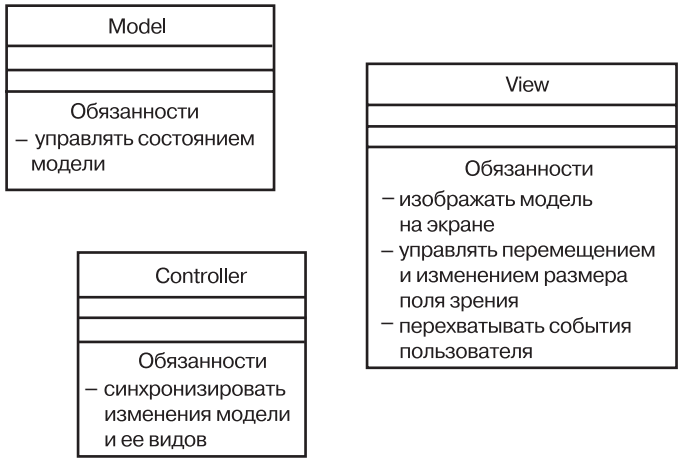


Рис. 4.10. Моделирование распределения обязанностей в системе

- Рассмотреть способы взаимодействия классов и перераспределить их обязанности исходя из того, что ни один класс не должен делать слишком мало или слишком много.

Набор классов формирует образец (см. главу 29).

В качестве примера на рис. 4.10 приведен набор классов, взятый из языка Smalltalk, показывающий распределение обязанностей между классами `Model` (Модель), `View` (Представление) и `Controller` (Контроллер). Все эти классы работают вместе так, что ни одному из них не приходится делать слишком много или мало.

Моделирование непрограммных сущностей

Иногда сущности, которые вы моделируете, не имеют аналога в программном обеспечении. Например, люди, которые посылают счета, и роботы, которые автоматически пакуют заказы для поставок со складов, могут быть частью *потока работ* (workflow), моделируемого для торговой системы. Ваше приложение может не иметь никаких программных компонентов, которые представляют сущности (в отличие от заказчиков в приведенном примере, поскольку ваша система, вероятно, будет поддерживать информацию о них).

Чтобы моделировать непрограммные сущности, необходимо:

- Смоделировать абстрагируемые сущности в виде классов.

Стереотипы обсуждаются в главе 6.

- Если вы хотите выделить непрограммные сущности из определенных в UML стандартных строительных блоков, создайте новый строительный блок, применив стереотипы для спецификации этой новой семантики и закрепив за ним некий отличительный символ.

Узлы обсуждаются в главе 27.

- Если моделируемая сущность представляет собой часть аппаратного обеспечения, которая, в свою очередь, содержит программное обеспечение, рассмотрите ее моделирование в виде узла определенного рода – таким образом, чтобы позднее можно было расширить представление ее структуры.

На заметку. Язык UML в основном предназначен для моделирования программных систем, хотя в сочетании с текстовыми языками моделирования аппаратного обеспечения – такими, например, как VHDL – он может быть достаточно выразительным для моделирования аппаратной части. Более того, консорциум OMG предложил расширение UML под названием SysML, предназначенное для моделирования систем.

Внешние сущности часто моделируются как действующие лица (см. главу 17).

Как показано на рис. 4.11, вполне допустимо абстрагировать людей (в данном примере – AccountsReceivableAgent, АгентПоДебиторскойЗадолженности) и аппаратное обеспечение (Robot – Робот) в виде классов, потому что они представляют собой множества объектов с общей структурой и общим поведением.

Рис. 4.11. Моделирование непрограммных сущностей

Моделирование примитивных типов

Типы обсуждаются в главе 11.

В противоположность сказанному выше, иногда моделируемые сущности могут браться непосредственно из языка программирования, который используется для реализации решения. Как правило, эти абстракции включают примитивные типы – такие как целые числа, символы, строки и даже перечислимые типы, созданные вами.

Чтобы моделировать примитивные типы, необходимо:

- Смоделировать абстрагируемую сущность в виде класса или *перечисления* (enumeration), изображая ее в нотации класса с соответствующим стереотипом.

Ограничения обсуждаются в главе 6.

- Если вам нужно специфицировать диапазон значений, ассоциированный с этим типом, используйте *ограничения* (constraints).

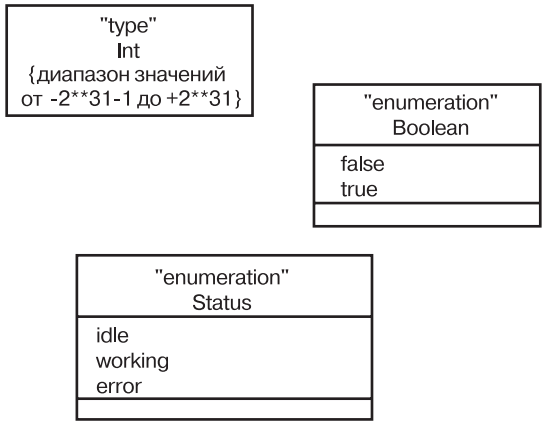


Рис. 4.12. Моделирование примитивных типов

По рис. 4.12 видно, что эти сущности могут быть смоделированы на UML как типы или перечисления, которые изображаются как классы, но с явно отмеченными стереотипами. Примитивные типы, такие как целые числа (представленные классом Int), моделируются как типы, и вы можете явно указать диапазоны их допустимых значений, используя ограничения; семантика примитивных типов должна быть определена вне UML. Перечислимые типы, скажем Boolean (Булев)

и Status (Состояние) могут быть смоделированы перечислениями со своими собственными индивидуальными литералами, указанными в разделе атрибутов (отметим, однако, что атрибутами они не являются). Перечислимые типы также могут определять операции.

На заметку. Некоторые языки, в частности C и C++, позволяют устанавливать целые значения для литералов перечислений. Вы можете смоделировать эти значения на UML, добавив комментарий к описанию литерала перечисления. Однако для логического моделирования целые значения не нужны.

Советы и подсказки

Моделируя классы на UML, помните, что каждый класс должен отображать некоторую осязаемую или концептуальную абстракцию из предметной области конечного пользователя или разработчика. Хорошо структурированный класс обладает следующими признаками:

- представляет четкую абстракцию понятия из словаря проблемной области или области решения;
- включает краткий, четко выраженный набор обязанностей и справляется с ними наилучшим образом;
- представляет четкое разделение спецификации и реализации этой абстракции;
- понятен и прост, но при этом является расширяемым и адаптируемым.

Изображая класс в UML, вы должны:

- показать только те его свойства, которые важны для понимания абстракции в данном контексте;
- организовать длинные списки атрибутов и операций, группируя их по категориям;
- представить взаимосвязанные классы на одной диаграмме.

Глава 5. Связи

В этой главе:

Научившись строить абстракции, вы быстро замечаете, что очень немногие классы существуют сами по себе. Наоборот, большинство из них кооперируется с другими самыми разными способоами. Поэтому при моделировании системы вы не только должны идентифицировать сущности, формирующие ее словарь, но и смоделировать отношения их друг к другу.

Более развитые виды связей обсуждаются в главе 10.

В объектно-ориентированном моделировании существуют три вида связей между классами, которые наиболее важны: *зависимость*, представляющая связи использования между классами (включая уточнение, трассировку и связывание); *обобщение*, которое связывает обобщенные классы с их специализациями; *ассоциации*, описывающие структурные связи объектов. Каждая из этих разновидностей представляет отдельный способ комбинирования абстракций.

Построение сетей связей не сильно отличается от создания сбалансированного распределения обязанностей между классами: если вы усложните их, то запутанный клубок связей сделает вашу модель непонятной; если же чрезмерно упростите, то лишитесь всего того богатства возможностей, которое обеспечила бы вашей системе кооперация.

Введение

Вернемся к метафоре строительства дома. Стены, двери, окна, встроенные шкафы и системы освещения формируют часть вашего словаря. Однако ни одна их этих вещей не существует в отдельности: одна стена соединяется с другими, двери и окна вмонтированы в стены, открывая в них проемы, встроенные шкафы и системы освещения крепятся к стенам и потолкам. И все эти объекты – стены, двери, окна, шкафы и осветительные приборы – в совокупности формируют более высокоуровневые сущности (комнаты).

Тем не менее вы обнаружите между этими сущностями не только структурные, но и иные связи. Например, ваш дом имеет окна, но они могут быть неодинаковы. У вас может быть большое окно в гостиной, которое не открывается, и маленькое кухонное, которое открывается. Некоторые окна открываются вверх или вниз; другие (скажем, патио) сдвигаются влево или вправо. Далее, рама бывает одинарной или двойной и т.д. Но независимо от этих различий, в каждом окне присутствует некоторый признак «оконности»: все они открывают проем в стене и предназначены для того, чтобы пропускать свет, воздух, а иногда даже людей.

В UML способы соединения сущностей друг с другом, логические либо физические, моделируются *связями*. В объектно-ориентированном моделировании существует три типа наиболее важных связей: зависимости, обобщения и ассоциации.

1. *Зависимость* представляет собой связь использования. Например, трубы зависят от водонагревателя для подогрева воды, которая по ним передается.
2. *Ассоциация* – это структурная связь между экземплярами. Например, комнаты состоят из стен и других объектов; в стены вмонтированы двери и, возможно, окна; через стены могут тянуться трубы.
3. *Обобщение* связывает обобщенные классы с более специализированными и потому известны как связи наследования («класс-подкласс», или «родитель-потомок»). Например, витраж – это окно с очень большими, жестко фиксированными панелями; патио – разновидность окна, открывающегося вбок.

Другие виды связей, такие как реализация и уточнение, обсуждаются в главе 10.

Эти три вида связей описывают большинство основных способов взаимодействия сущностей. Неудивительно, что они хорошо проецируются и на те способы, которые предусмотрены большинством объектно-ориентированных языков для соединения объектов.

Как показано на рис. 5.1, UML предлагает особое графическое представление для каждого вида связи. Эта нотация позволяет

визуализировать связи независимо от конкретного языка программирования, причем способом, описывающим наиболее важные параметры связей: имя, соединяемые сущности и свойства.

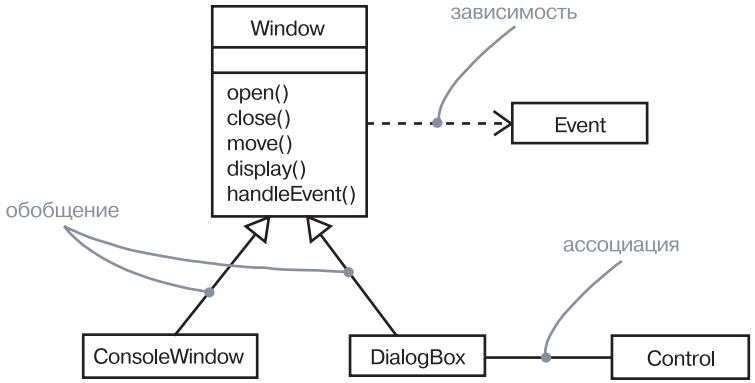


Рис. 5.1. Связи

Базовые понятия

Связь (relationship) – это соединение сущностей. В объектно-ориентированном моделировании есть три наиболее важных вида связей: зависимости, обобщения и ассоциации. Связь изображается в виде пути с использованием разнообразных типов линий, каждый из которых соответствует определенному виду связи.

Зависимости

Зависимость (dependency) – это связь, которая устанавливает, что одна сущность, например класс Window (Окно), использует информацию и сервис (операцию либо услугу), представляемые другой сущностью, например классом Event (Событие), но не обязательно – наоборот. Зависимость изображается в виде пунктирной линии со стрелкой, направленной на зависимую сущность. Выбирайте зависимость, когда вам нужно показать, что одна сущность использует другую.

Примечания обсуждаются в главе 6, пакеты – в главе 12.

Чаще всего вы будете применять зависимости для того, чтобы показать, что один класс использует операции другого класса (либо использует переменные или аргументы типа другого класса) – см. рис. 5.2. Это очень похоже на связь использования: если используемый класс изменяется, это может затронуть операции других классов, поскольку используемый класс теперь может предоставлять другой интерфейс или поведение. UML допускает создание зависимостей и между другими элементами, в частности между примечаниями и пакетами.

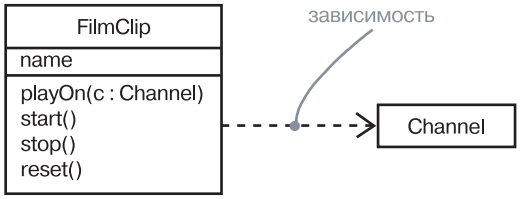


Рис. 5.2. Зависимости

Разные формы зависимостей обсуждаются в главе 10, стереотипы – в главе 6.

На заметку. Зависимость может быть поименована, хотя такие имена редко требуются на практике, если только вы не моделируете слишком много зависимостей, которые иначе было бы трудно отличать друг от друга. Чаще для того, чтобы подчеркнуть различия между зависимостями, используют стереотипы.

Обобщения

Обобщение (generalization) – это связь между сущностью общего характера (называемой суперклассом, или родителем) и более специфичной сущностью (называемой подклассом, дочерним классом или потомком). Иногда обобщение называют связью типа «является»: к примеру, класс `BayWindow` (Окно-«фонарь») является разновидностью более общей сущности, класса `Window` (Окно). Объекты дочернего класса могут быть использованы как переменные или параметры типа его родителя, но не наоборот. Другими словами, дочерняя сущность может быть подставлена там, где объявлена родительская. Дочерняя сущность наследует свойства родителя, а именно его атрибуты и операции. Часто, хотя и не всегда, потомок имеет дополнительные атрибуты и операции помимо родительских. Реализация операции в дочернем классе замещает реализацию той же операции родителя – это явление называется *полиморфизмом*. Одинаковые операции должны иметь одинаковую *сигнатуру* (имя и параметры).

Графически обобщение представлено сплошной линией со стрелкой в форме большого пустого треугольника, указывающей на родителя (см. рис. 5.3). Используйте обобщение, когда необходимо изобразить связь «родитель-потомок».

Класс может одного или нескольких родителей либо не иметь их вовсе. Класс, не имеющий родителей, но имеющий одного или нескольких потомков, называется *корневым* (root) или *базовым*. Класс, не имеющий потомков, называется *листовым* (leaf). О классе, у которого есть только один родитель, говорят, что он использует *одиночное наследование*, в отличие от класса, у которого более чем один родитель (*множественное наследование*).

Пакеты обсуждаются в главе 12.

Чаще всего вы будете использовать обобщения между классами и интерфейсами, чтобы отобразить связь наследования. Кроме того, в UML можно описать обобщение между другими элементами, например узлами.

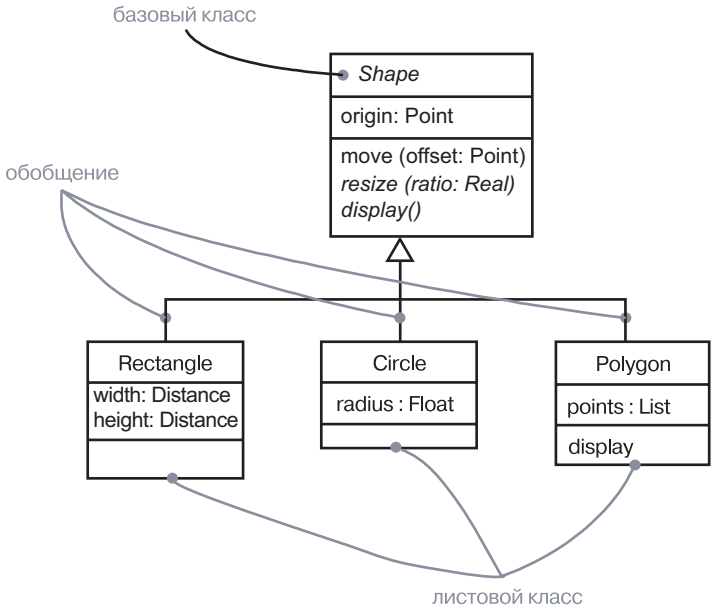


Рис. 5.3. Обобщение

На заметку. Обобщение с именем означает декомпозицию суперкласса на подклассы в каком-то определенном аспекте, называемом *набором обобщения* (generalization set). Множественные наборы обобщения ортогональны; суперкласс может быть специализирован с применением множественного наследования для выбора одного подкласса из каждого набора обобщения. Это отдельная тема, которую мы не планировали раскрывать в рамках данной книги.

Ассоциации и зависимости (но не обобщения) могут быть рефлексивными (см. главу 10).

Ассоциации

Ассоциация – это структурная связь, указывающая, что объекты одной сущности соединяются с объектами другой. Так, имея ассоциацию между двумя классами, вы можете соединить объекты одного класса с объектами другого. Вполне допустимо, чтобы оба конца ассоциации соединяли один и тот же класс – иными словами, один объект класса может связываться с другим объектом того же класса. Ассоциация, связывающая два класса, называется *бинарной*.

Хоть и нечасто, но все же встречаются так называемые *n-арные ассоциации*, которые соединяют более двух классов .

Графически ассоциация представлена сплошной линией, два конца которой соединяют один или разные классы. Применяйте ассоциацию, когда хотите показать структурные связи.

Выше мы описали базовый вариант. Но существуют еще пять дополнений, применяемых к ассоциациям.

Не путайте направление имени с навигацией по ассоциации (см. главу 10).

Ассоциация может иметь **имя**, используемое для описания природы **связи**. Поэтому значение имени не должно быть двусмысленным. Используя стрелочку в форме треугольника, вы можете указать направление, в котором следует читать это имя (см. рис. 5.4).

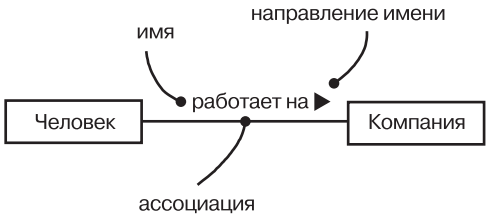


Рис. 5.4. Имена ассоциации

На заметку. Хотя ассоциация может быть поименована, обычно нет необходимости указывать ее имя, если указаны конечные имена ассоциации. Если в вашей модели присутствует несколько ассоциаций, связывающих одни и те же классы, необходимо использовать или имена ассоциаций, или конечные имена, чтобы различать их. Если ассоциация имеет несколько концов для одного и того же класса, требуется в обязательном порядке указывать ее конечные имена, чтобы различать эти концы. При наличии только одной ассоциации, соединяющей пару классов, некоторые авторы моделей опускают имена, но все же лучше указать их, чтобы прояснить назначение ассоциации.

Роли имеют отношение к семантике интерфейсов, как показано в главе 11.

Когда класс участвует в ассоциации, он выполняет в этой связи конкретную роль. **Роль** – это «лицо» класса, который находится на дальнем конце ассоциации, представленное классу, находящемуся на ее ближнем конце. Вы можете явно именовать роль, которую выполняет класс в ассоциации.

Роль, которую играет класс, находящийся на конце ассоциации, называется *конечным именем* (в первой версии UML она называлась *именем роли*). На рис. 5.5 класс `Person` (Человек), играющий роль `employee` (работник), ассоциирован с классом `Company` (Компания), играющим роль `employer` (работодатель).

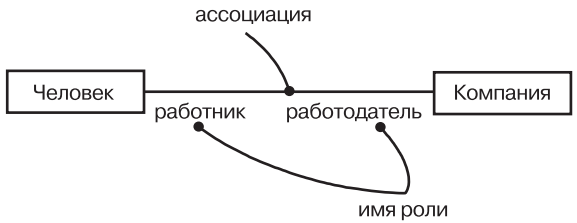


Рис. 5.5. Конечные имена ассоциации (имена ролей)

На заметку. Один и тот же класс может играть в разных ассоциациях одну и ту же роль или разные роли.

На заметку. Атрибут может рассматриваться как односторонняя ассоциация, принадлежащая классу. Имя атрибута соответствует конечному имени ассоциации вне класса.

Экземпляр ассоциации называется ссылкой (см. главу 16).

Ассоциация представляет структурную связь между объектами. Во многих ситуациях моделирования важно знать, сколько объектов может быть соединено одним экземпляром ассоциации. Этот параметр называется множественностью роли ассоциации. **Множественность** (multiplicity) представляет диапазон целых чисел, указывающий возможное количество связанных объектов. Он записывается в виде выражения с минимальным и максимальным значением, которые могут быть равны; для их разделения используются две точки. Устанавливая множественность дальнего конца ассоциации, вы указываете, сколько объектов может существовать на дальнем конце ассоциации для каждого объекта класса, находящегося на ближнем ее конце. Количество объектов должно находиться в пределах заданного диапазона. Множественность может быть определена как единица (1), ноль или один (0..1), много (0..*), один или несколько (1..*). Вы можете задавать диапазон целых значений, например 2..5, или устанавливать точное число, например 3 (эквивалент записи 3..3).

Рис. 5.6. Множественность

В примере на рис. 5.6 каждая компания может нанимать одного или нескольких человек (множественность 1..*); каждому человеку сопоставлено 0 или более компаний-работодателей (множественность * – эквивалент записи 0..*).

Существует несколько важных разновидностей агрегации. О них пойдет речь в главе 10.

Простая ассоциация между двумя классами представляет структурную связь между равноправными элементами: оба класса концептуально находятся на одном уровне – ни один из них не может считаться важнее другого. Рано или поздно вам понадобится смоделировать связь «целое-часть», в которой один класс представляет крупную сущность (целое), содержащую в себе более мелкие (части). Этот тип связей, основанных на отношениях *обладания*, называется **агрегацией** и подразумевает, что объект-целое обладает объектами-частями. По сути, агрегация – это особый вид ассоциации, поэтому изображается она линией простой ассоциации, к которой добавлен пустой ромбик со стороны объекта-целого (см. рис. 5.7).

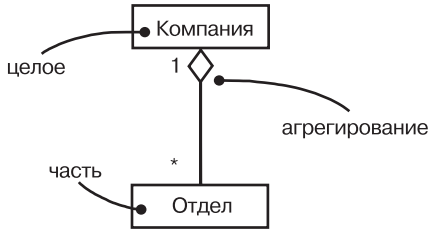


Рис. 5.7. Агрегация

На заметку. Простая форма агрегации выглядит так не случайно: пустой ромбик отличает целое от части – ни больше ни меньше. Это означает, что простая агрегация не изменяет смысла навигации по ассоциации между целым и его частями. Более строгая форма агрегации описывается в разделе «Ассоциации» главы 10.

Прочие свойства

Сложные понятия связей обсуждаются в главе 10.

Простые зависимости, обобщения и ассоциации с именами, показателями множественности и ролями – это наиболее общие средства, с которыми вы имеете дело, создавая абстракции. Основных форм этих трех абстракций достаточно, чтобы выразить наиболее важную семантику связей в большинстве моделей. Однако иногда вам нужно будет визуализировать или специфицировать другие свойства – такие как композитная агрегация, навигация, дискриминанты, ассоциации-классы, некоторые разновидности зависимостей и обобщений. Эти и многие другие свойства можно выразить на UML, но обычно их трактуют как более сложные и, соответственно, менее употребительные понятия.

Диаграммы классов рассматриваются в главе 6. Ссылки обсуждаются в главе 16.

Зависимости, обобщения и ассоциации являются *статическими* сущностями, определенными на уровне классов. В UML эти связи обычно визуализируются в диаграммах классов.

Начиная моделировать на уровне объектов (особенно когда приходится работать с их динамическими кооперациями), вы сталкиваетесь со ссылками – экземплярами ассоциаций, представляющими соединения между объектами, по которым можно отправлять сообщения.

Стили изображения

Связи изображаются на диаграммах линиями, соединяющими одни пиктограммы с другими. Для различения типов связей к этим линиям добавляются разнообразные элементы, например стрелки или ромбики. Как правило, авторы моделей выбирают один из двух стилей рисования линий:

- *наклонные линии под любым углом.* Используются линии из одного сегмента (если только не нужно несколько, чтобы миновать другие пиктограммы);
- *линии, образующие квадрат и проведенные параллельно краям страницы.* Если только линии не соединяют две выровненные пиктограммы, они должны быть прорисованы строго по горизонтали и вертикали, соединяясь друг с другом под прямым углом. В данной книге главным образом используется именно этот стиль.

Если постараться, пересечения линий в большинстве случаев можно избежать. Если оно все-таки неизбежно и есть опасность двусмысленной интерпретации связей, стоит отметить место пересечения линий маленьким полукругом (рис. 5.8).

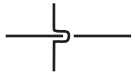


Рис. 5.8. Символ пересечения линий

Типичные приемы моделирования

Моделирование простых зависимостей

Часто встречающийся вид зависимости – это связь класса, который использует другой класс в качестве параметра своей операции. Чтобы смоделировать эту связь использования, необходимо создать зависимость, направленную от класса с операцией к классу, используемому в качестве ее параметра.

В примере на рис. 5.9 показан набор классов, составляющих систему, которая управляет назначением студентов и преподавателей на курсы в университете. Этот рисунок изображает зависимость от CourseSchedule (РасписаниеКурсов) к Course (Курс), поскольку Course используется в операциях add (добавить) и remove (удалить) класса CourseSchedule.

Если вы представите полную сигнатуру операций, подобно тому как это сделано на рисунке, вам, скорее всего, необязательно показывать эту зависимость, потому что использование класса уже присутствует в сигнатуре. Однако иногда показать такую зависимость необходимо, особенно если вы не приводите сигнатуру операций или ваша модель показывает другие связи с используемым классом.

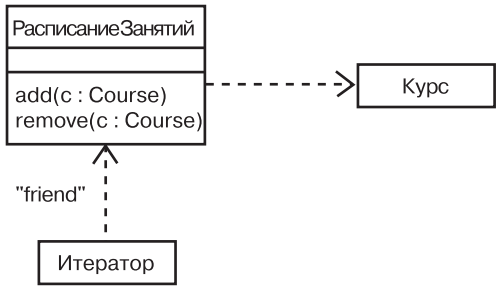


Рис. 5.9. Связи зависимости

Другие стереотипы связей обсуждаются в главе 16.

Кроме того, рис. 5.9 демонстрирует зависимость, которая не связана с классами в операциях, а моделирует одну общую идиому C++. Зависимость от Iterator (Итератор) говорит о том, что Iterator использует CourseSchedule, то есть CourseSchedule ничего не знает об Iterator. Зависимость помечена стереотипом <permit> (<разрешить>), который подобен предложению friend (друг) в C++.

Моделирование одиночного наследования

В моделировании словаря вашей системы вы часто будете сталкиваться с классами, которые структурно или по своему поведению похожи на другие классы. Каждый из них можно смоделировать как отдельную независимую абстракцию. Но лучше извлечь из них общие структурные или поведенческие части и поместить их в более общие классы, от которых эти классы будут унаследованы.

Чтобы смоделировать связи наследования, необходимо:

- Найти в наборе классов обязанности, атрибуты и операции, общие для нескольких классов.

- Перенести эти обязанности, атрибуты и операции в более общий класс. Если необходимо, создать такой класс (но будьте осторожны – не создавайте слишком много уровней).
- Назначить более специализированные классы потомками общих классов, проведя связь обобщения от каждого специализированного класса к его родителю.

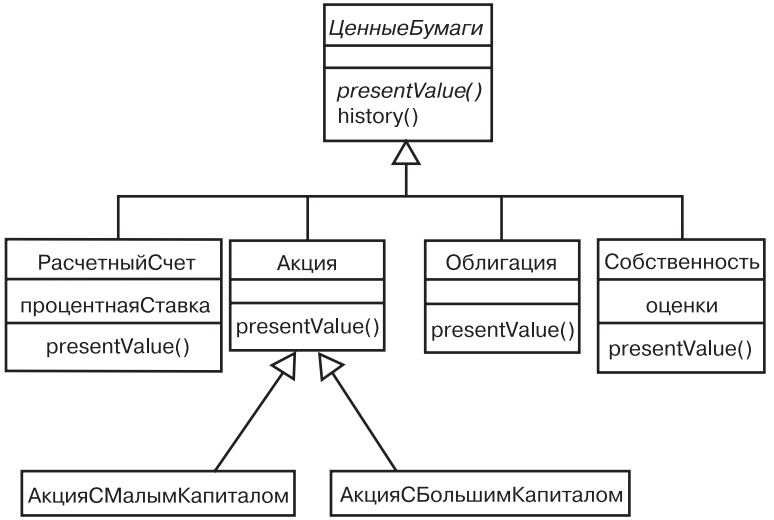


Рис. 5.10. Связи наследования

На рис. 5.10 представлен ряд классов, формирующих приложение управления торговлей. Вы найдете здесь связь обобщения, проведенную от четырех классов – CashAccount (РасчетныйСчет), Stock (Акция), Bond (Облигация) и Property (Собственность), – к более общему классу Security (ЦенныеБумаги). Класс Security является родителем, а CashAccount, Stock, Bond и Property – потомками (дочерними классами). Каждый из этих четырех специализированных классов представляет собой некоторую разновидность Security. Последний включает две операции: presentValue (текущаяСтоимость) и history (История). Поскольку Security является родителем CashAccount, Stock, Bond и Property, все они наследуют эти две операции, равно как и любые другие атрибуты и операции Security, которые могут быть не указаны на этом рисунке.

Абстрактные классы и операции обсуждаются в главе 9.

Имена Security и presentValue написаны курсивом не случайно. Когда вы строите иерархии, подобные той, что представлена на рис. 5.10, то часто сталкиваетесь с нелистовыми классами, которые неполны или для которых не может существовать объектов. Такие классы называются *абстрактными*. В UML вы можете пометить класс как абстрактный, написав его имя курсивом. То же касается и операций: в рассматриваемом примере операция presentValue имеет сигнатуру,

но не реализуется в классе Security и должна быть реализована в виде методов на более низком уровне абстракции, поэтому ее имя также выделено курсивом. Зато все четыре непосредственных потомка Security являются конкретными (то есть не абстрактными), а значит, каждый из них обязан представить конкретную реализацию операции presentValue.

Иерархии обобщения/специализации не обязательно ограничиваются двумя уровнями. Обычно этих уровней больше, и рис. 5.10 служит тому подтверждением. SmallCapStock (АкцияСМалымКапиталом) и LargeCapStock (АкцияСБольшимКапиталом) – потомки класса Stock, который, в свою очередь, является дочерним по отношению к Security. Таким образом, Security – корневой класс, поскольку он не имеет родителей. SmallCapStock и LargeCapStock – листовые классы, потому что не имеют потомков. Stock имеет как родителя, так и потомков; он не подходит под определение ни корневого, ни листового класса.

Множественное наследование обсуждается в главе 10.

Хотя это и не показано здесь, вы можете создавать классы с несколькими родителями. Это называется множественным наследованием и означает, что классу-потомку передаются все атрибуты, операции и ассоциации его родителей.

Конечно, в системе наследования не может быть циклов: ни один класс не может быть своим собственным предком.

Моделирование структурных связей

При использовании связей зависимости или обобщения вы можете моделировать классы, которые находятся на разных уровнях важности либо разных уровнях абстракции. В случае зависимости между двумя классами один класс зависит от другого, но этот последний ничего не знает о первом. В случае же обобщения между классами потомок наследует свойства родителя, но родитель не содержит никаких конкретных сведений о своих потомках. Иными словами, связи зависимости и обобщения *асимметричны*.

Ассоциации по умолчанию двунаправленны, но вы можете ограничить их направленность (см. главу 10).

Моделируя ассоциации, вы имеете дело с классами, находящимися на одном уровне. Два класса, вовлеченные в ассоциацию, некоторым образом связаны друг с другом, и часто вы можете осуществлять навигацию по этой связи в любом направлении.

В то время как зависимость – связь использования, а обобщение – связь наследования, ассоциация определяет структурный путь взаимодействия объектов двух классов.

Чтобы смоделировать структурные связи, воспользуйтесь следующими рекомендациями:

- Установите ассоциацию между каждой парой классов, по объектам которых следует обеспечить навигацию. Это основанное на данных представление ассоциации.

- Установите ассоциацию между каждой парой классов, объекты одного из которых нуждаются во взаимодействии с объектами другого (но не в виде локальных переменных в процедуре или параметров операции). Это в основном поведенческое представление ассоциации.
- Для каждой из установленных ассоциаций специфицируйте множественность, особенно когда она не выражается символом * (звездочка), принятым по умолчанию, а также имена ролей, особенно если они помогают объяснить модель.
- Если класс на одном конце ассоциации структурно или организационно представляет целое, в то время как класс, находящийся на другом конце, является частью этого целого, пометьте такую связь как агрегацию (то есть добавьте к связывающей их линии ромбик со стороны целого).

Варианты использования обсуждаются в главе 17.

Как вы можете узнать, когда объекты одного класса должны взаимодействовать с объектами другого? Ответ – когда CRC-карточки и анализ на основе вариантов использования заставляют вас рассмотреть структурные и поведенческие сценарии. Если вы обнаруживали, что какие-либо классы взаимодействуют, используя связи по данным, установите между ними ассоциацию.

Рис. 5.11 показывает набор классов информационной системы учебного заведения. Перечислим эти классы начиная с нижнего левого угла диаграммы: Student (Студент), Course (Курс) и Instructor (Преподаватель). Существует ассоциация между Student и Course, свидетельствующая о том, что студенты посещают курсы. Более того, каждый студент может посещать многие курсы и каждый курс может быть прочитан для многих студентов. Также вы обнаружите ассоциацию между Course и Instructor, которая говорит о том, что преподаватели ведут курсы. Для каждого курса назначен как минимум один преподаватель, причем каждый преподаватель может вести один или несколько курсов либо не вести ни одного. Каждый курс относится исключительно к одному факультету.

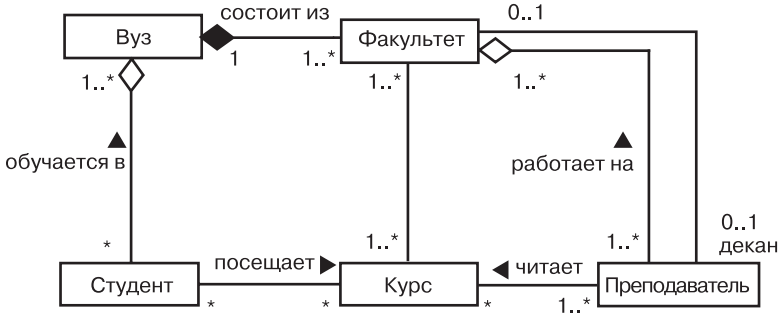


Рис. 5.11. Структурные связи

Агрегация между классами `School` *и* `Department` *представляет собой композицию. Композиция – это более строгая форма агрегации, подразумевающая монопольное владение (см. главу 10).*

Связи между классом `School` (Учебное заведение) и классами `Student` и `Department` (Факультет) немного иные. Здесь вы видите примеры агрегаций. Вуз может набрать или не набрать студентов; каждый студент может быть зарегистрирован в одном или нескольких вузах; в вузе имеется один или несколько факультетов, причем каждый относится исключительно к одному учебному заведению. Вы можете опустить дополнительные символы агрегации и применять простые ассоциации, но, специфицируя `School` как целое, а `Student` и `Department` – как его части, вы тем самым проясняете, что по отношению к чему является главным. По рисунку видно, что учебные заведения в каком-то смысле определяются его студентами и факультетами. Впрочем, студенты и факультеты тоже не существуют сами по себе, вне вуза, к которому они прикреплены. Они получают свой статус только в связи с учебным заведением (так, человек, не учащийся в вузе, лишается права именоваться студентом).

Кроме того, существуют две ассоциации между `Department` и `Instructor`. Одна из них показывает, что каждый преподаватель работает на одном или нескольких факультетах и каждому факультету сопоставлен один или несколько преподавателей. Эта связь моделируется как агрегация, поскольку организационно факультеты в структуре учебного заведения находятся выше, чем преподаватели. Вторая ассоциация, установленная для каждого факультета, говорит о том, что на факультете есть один начальник – преподаватель, занимающий должность декана. Из приведенной модели явствует, что преподаватель может быть деканом не более чем на одном факультете, а некоторые преподаватели не являются деканами ни одного факультета.

На заметку. Модель, приведенная на рис. 5.11, не является единственно приемлемой. К примеру, учебное заведение может не иметь факультетов. Декан не всегда является преподавателем, а иногда даже встречаются преподаватели-студенты. Все это не значит, что рассматриваемая нами модель не идеальна, она просто представляет один из возможных вариантов организационной структуры вуза. Каждая ваша модель, подобная этой, будет строиться исходя из конкретных условий.

Советы и подсказки

Моделируя связи на UML, соблюдайте следующие правила:

- используйте зависимости только тогда, когда моделируемые связи не являются структурными;

- используйте обобщение, только когда оно выражает связь типа «является». Множественное наследование часто может быть заменено агрегацией;
- избегайте циклических связей обобщения;
- старайтесь обеспечить сбалансированность связей обобщения. Цепочки наследования не должны быть ни слишком длинными (глубина более четырех уровней уже нежелательна), ни слишком широкими (ищите возможность создания промежуточных абстрактных классов);
- при наличии структурных связей между объектами используйте главным образом ассоциации. Но не применяйте их для того, чтобы выразить временные связи, например в параметрах или локальных переменных процедур.

Представляя связи в UML графически, не забывайте о следующем:

- согласуйте в модели прямоугольные и наклонные линии. Прямоугольные линии визуально выражают соединение связанных сущностей с одним общим родителем. Наклонные линии часто позволяют сэкономить пространство на сложных диаграммах. Применение линий обоих типов на одной диаграмме удобно для привлечения внимания к разным группам связей;
- избегайте пересекающихся линий там, где можно без них обойтись;
- показывайте только те связи, которые нужны для понимания определенной группы сущностей. Излишние связи (особенно избыточные ассоциации) нежелательны.

Глава 6. Общие механизмы

В этой главе:

- Примечания
- Стереотипы, помеченные значения и ограничения
- Моделирование комментариев
- Моделирование новых строительных блоков
- Моделирование новых свойств
- Моделирование новой семантики
- Расширения UML

Общие механизмы также обсуждаются в главе 2.

Работа с UML значительно упрощается благодаря наличию четырех общих механизмов, которые используются в языке повсеместно: спецификаций, дополнений, общих средств разделения и средств расширения. Настоящая глава посвящена двум из них: дополнениям и средствам расширения.

Примечание – наиболее важное из дополнений, которое может существовать и обособленно. Примечание представляет собой графический символ для описания ограничений или комментариев, присоединенных к элементу или набору элементов. Он пригодится вам для того, чтобы включить в модель такую информацию, как требования, замечания, оценки и пояснения.

Механизмы расширений UML (стереотипы, помеченные значения и ограничения) обеспечивают возможность расширять этот язык строго контролируемым образом. Стереотипы расширяют словарь UML, позволяя создавать новые строительные блоки, производные от существующих, но при этом «подогнанные» специально под вашу проблемную область. Помеченные значения расширяют свойства стереотипов UML, позволяя описывать новую информацию в спецификации данного элемента. Ограничения расширяют семантику строительных блоков UML, позволяя вам добавлять новые правила или модифицировать существующие. Все эти механизмы помогут вам адаптировать UML для вашей области и методики работы.

Введение

Иногда в модель нужно включить информацию, выходящую за рамки обычных формальностей. Так, архитектор порой пишет примечания на комплекте чертежей здания, чтобы обратить внимание строителей на некоторые тонкости. В студии звукозаписи композитор может изобрести новую музыкальную нотацию, чтобы выразить с ее помощью некоторые необычные эффекты, требуемые от исполнителя. Хотя в обоих случаях существуют хорошо определенные языки – язык строительных чертежей и язык музыкальной нотации, – их приходится некоторым образом модифицировать или расширять, чтобы наиболее адекватно выразить ваши намерения.

Моделирование – это обмен информацией. UML представляет полный набор инструментов для визуализации, специфицирования, конструирования и документирования артефактов разнообразных программных систем, однако при некоторых обстоятельствах возникает необходимость в его модификации и расширении. Такие процессы постоянно протекают в человеческих языках (вот почему каждый год публикуются новые словари), поскольку ни один язык не может быть достаточным на все времена для передачи всего, что вы хотите выразить. UML вы тоже используете в целях коммуникации, а значит, вам нужно будет придерживаться его основных принципов и выразительных средств, если только не возникнет достаточно веских причин, чтобы от них отступить. И когда вы обнаружите необходимость отказаться от обычных формальных средств языка и прибегнуть к чему-то новому, то должны действовать строго определенным образом. В противном случае то, что вы делаете, никто не поймет адекватно.

Примечания – это механизм, предусмотренный в UML для того, чтобы включить в модель произвольные комментарии и ограничения, поясняющие ее. Примечания могут представлять собой артефакты, играющие важную роль в жизненном цикле разработки программного обеспечения (например, требования) либо обзоры или пояснения, выраженные в свободной форме.

UML предусматривает графическое представление комментариев и ограничений, называемых примечаниями (рис. 6.1). Эта нотация позволяет вам размещать комментарии прямо на диаграмме. В сочетании с правильными инструментами примечания также дают вам возможность указать ссылки либо непосредственно встроить в модель другие документы.

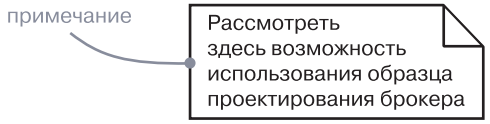


Рис. 6.1. Примечания

Стереотипы, помеченные значения и ограничения являются механизмами UML, предназначенными для добавления новых строительных блоков, создания новых свойств и специфицирования новой семантики. Например, если вы моделируете сеть, вам могут понадобиться символы для обозначения маршрутизаторов и концентраторов; вы можете использовать узлы со стереотипами для того, чтобы сделать эти элементы примитивными строительными блоками. Другая ситуация: если вы – член команды, которая реализует проект, отвечающий за сборку, тестирование и поставку версий, то вам понадобится отслеживать номера версий и результаты тестов для каждой из основных подсистем. Для включения этой информации в ваши модели пригодятся помеченные значения. И наконец, если вы моделируете сложные системы реального времени, некоторые модели надо будет дополнить информацией о временных бюджетах и предельных сроках. Для формулировки требований, касающихся времени, подойдут ограничения.

UML предусматривает текстовое представление стереотипов, помеченных значений и ограничений (см. рис. 6.2). Стереотипы также позволяют вам вводить новые графические символы, позволяя визуально подчеркнуть тот факт, что ваши модели «говорят» на языке специфической предметной области или принятой у вас культуры разработки.

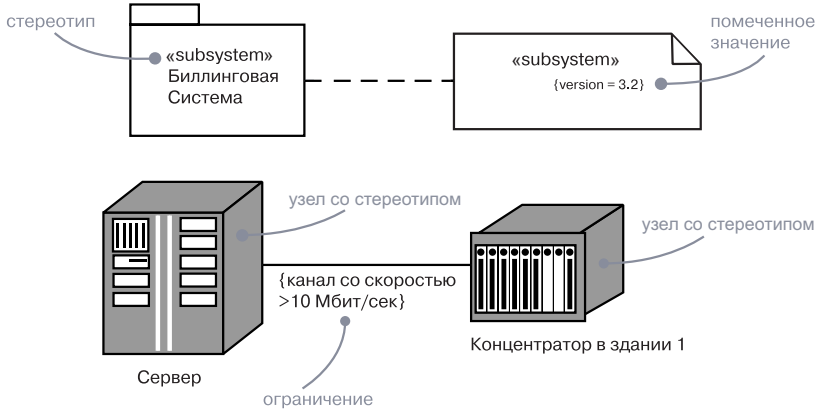


Рис. 6.2. Стереотипы, помеченные значения и ограничения

Базовые понятия

Примечание (note) – это графический символ для выражения ограничений или комментариев, присоединенных к элементу или набору элементов. Примечание изображается в виде прямоугольника с одним загнутым углом, включающего текстовый или графический комментарий.

Стереотип (stereotype) – это расширение словаря UML, позволяющее создавать новые виды строительных блоков, подобных существующим, но специфичных для выбранной проблемной области. Стереотип представлен именем, заключенным в угловые кавычки и помещенным над именем другого элемента. Элементы со стереотипом могут изображаться новой пиктограммой, ассоциированной с этим стереотипом.

Помеченное значение (tagged value) – это свойство стереотипа, позволяющее вводить новую информацию для относящихся к нему элементов. Представлено строкой в форме `имя = значение` внутри примечания, присоединенного к объекту.

Ограничение (constraint) – это текстовая спецификация семантики элемента UML, обеспечивающая добавление новых правил или модификацию существующих. Представлено строкой, заключенной в фигурные скобки и помещенной рядом с ассоциированным элементом либо связанной с ним зависимостью. В качестве альтернативы вы можете изображать ограничение, как примечание.

Примечания

Примечание, содержащее комментарий, не имеет семантического влияния, то есть его содержимое не изменяет смысла модели, в которую оно включено. Вот почему примечания используются для специфицирования требований, замечаний, оценок и пояснений наряду с выражением ограничений.

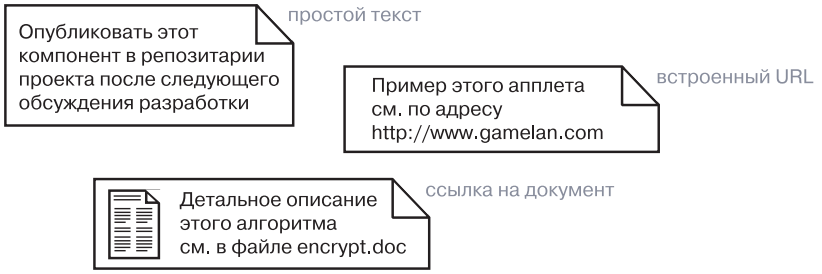


Рис. 6.3. Примечания

Посредством зависимостей примечания могут быть привязаны к нескольким элементам (см. главу 5).

Примечание может содержать текст и графику в любом сочетании. Если того требует ваш проект, вы можете разместить внутри комментария URL-адрес или ссылку на другой документ и даже встроить сам документ. Таким образом, вы можете использовать UML для организации всех артефактов, которые могут быть сгенерированы или использованы в процессе разработки (см. пример на рис. 6.3).

Прочие дополнения

Базовая нотация для ассоциаций вместе с некоторыми ее дополнениями обсуждается в главах 5 и 10.

Дополнения (adornments) – это текстовые или графические объекты, которые добавляются к базовой нотации элемента и используются для визуализации деталей его спецификации. Например, базовая нотация для ассоциации – линия, но она может быть снабжена дополнениями, описывающими роль и множественность каждого конца. Общее правило в UML таково: начинайте с базовой нотации каждого элемента, а затем включайте в модель дополнения, только если они необходимы для передачи существенно важной информации.

Большинство дополнений представлены текстом, который размещен возле поясняемого элемента, либо графическим символом, добавленным к базовой нотации. Однако иногда необходимо описать элемент более подробно, нежели этого можно достичь текстовыми или графическими средствами. Тогда, если речь идет о классах, компонентах и узлах, вы размещаете дополнительные разделы непосредственно под обычными (см. рис. 6.4).

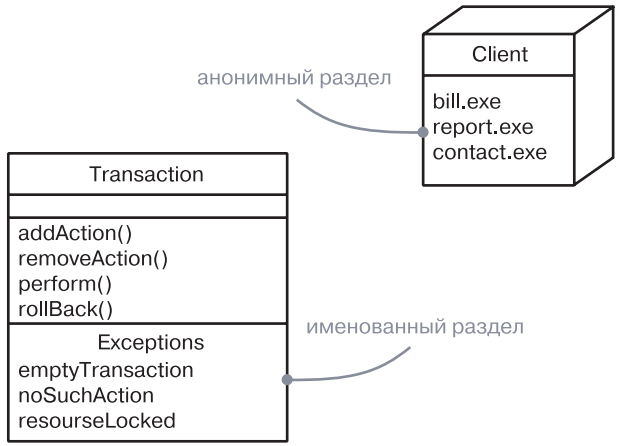


Рис. 6.4. Дополнительные разделы

На заметку. Полезный прием – давать имена всем дополнительным разделам, чтобы не возникало неясностей относительно их значения (если только содержимое дополнительного раздела не выражает его явно). Также желательно представлять дополнительные разделы как можно лаконичнее, поскольку злоупотребление ими загромождает диаграммы.

Стереотипы

Основные виды сущностей в UML обсуждаются в главе 2.

Технология Rational Unified Process описывается в приложении 2.

UML представляет собой язык описания структурных, поведенческих, группирующих и аннотирующих сущностей. Эти четыре базовых вида сущностей предназначены для выражения подавляющего большинства моделируемых систем. Однако иногда вам могут понадобиться другие сущности, соответствующие словарю вашей предметной области и выглядящие как примитивные строительные блоки.

Стереотип (stereotype) – это не то же самое, что родительский класс в обобщении «родитель-потомок». Скорее, можно рассматривать его как метатип (тип, определяющий другие типы), поскольку он создает эквивалент нового класса в метамодели UML. Например, при моделировании бизнес-процесса вы будете описывать исполнителей, документы и правила. Если вы следуете принципам такого процесса разработки, как Rational Unified Process, вам придется моделировать граничные (boundary) классы, управляющие (control) классы и классы-сущности (entities) – они опять же могут быть выражены стереотипами. Присваивая стереотип, скажем, узлу или классу, вы на самом деле расширяете UML за счет создания новых строительных блоков, подобных существующим, но с особыми моделирующими свойствами (каждый стереотип может представлять свой набор помеченных значений), семантикой (каждый стереотип может иметь собственные ограничения) и нотацией (каждый стереотип может обозначаться отдельной пиктограммой).

В своей простейшей форме стереотип изображен как имя, заключенное в угловые кавычки (например, «name») и помещенное над именем другого элемента. Для визуального представления стереотипа вы можете выбрать какую-либо пиктограмму и разместить ее справа от имени (если используется базовая нотация элемента) или же применить пиктограмму в качестве базового символа для отображения элементов, относящихся к стереотипу. Все эти варианты проиллюстрированы на рис. 6.5.

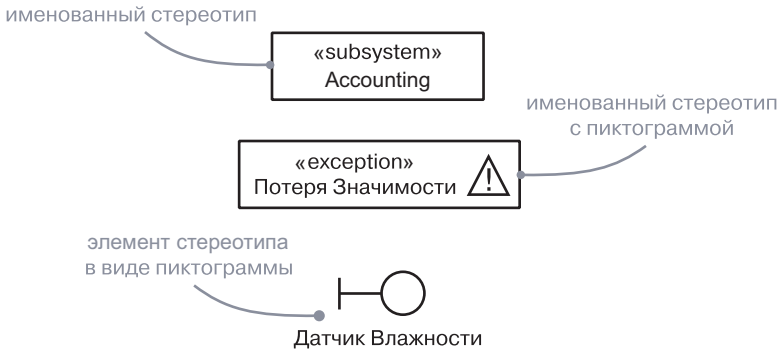


Рис. 6.5. Стереотипы

На заметку. Назначая стереотипу пиктограмму, попробуйте подчеркнуть некоторые нюансы с помощью цвета (но не увлекайтесь, иначе получится слишком пестро!) UML допускает использование пиктограмм любой формы, и если ваш проект того требует, они могут функционировать как примитивные инструменты (таким образом, пользователи, создающие диаграммы UML, получают в свое распоряжение палитру сущностей, которые отвечают их базовым потребностям и «говорят» на языке их проблемной области).

Помеченные значения

Каждый элемент в UML имеет собственный набор свойств: классы – имена, атрибуты и операции; ассоциации – имена и несколько концов, каждый из которых наделен своими особенностями, и т.д. Благодаря стереотипам вы можете привносить в UML новые сущности, а благодаря помеченным значениям – сообщать стереотипам новые свойства.

Атрибуты обсуждаются в главах 4 и 9.

Вы определяете теги, применяемые к индивидуальным стереотипам таким образом, что все внутри стереотипа имеет помеченное значение. *Помеченное значение* (tagged value) – не то же самое, что атрибут класса. Скорее, помеченные значения можно рассматривать как метаданные, поскольку они касаются спецификации элементов, а не их экземпляров. Так, например, вы можете указать требуемую производительность класса сервера либо потребовать, чтобы в системе использовался только определенный тип сервера (рис. 6.6).

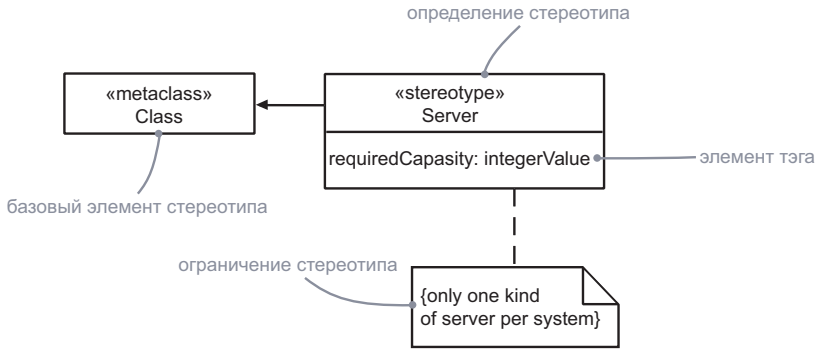


Рис. 6.6. Стереотипы и определения тегов

Помеченные значения входят в примечания, присоединенные к элементу, как показано на рис. 6.7. Каждое помеченное значение включает в себя строку, состоящую из имени (тега), разделителя (символа =) и значения тега.

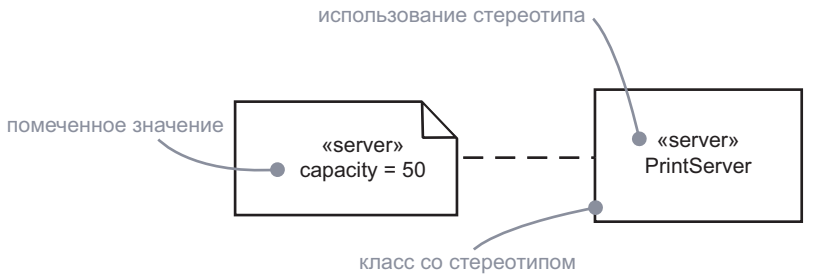


Рис. 6.7. Помеченные значения

На заметку. Один из наиболее частых случаев использования помеченных значений – определение свойств, важных для генерации кода или управления конфигурацией. Например, вы можете использовать помеченные значения для спецификации языка программирования, в котором реализуется определенный класс, а также для указания автора и версии какого-либо компонента.

Ограничения

Все элементы UML обладают определенной семантикой. Обобщения, как правило, воплощают принцип подстановки Лискова, а множественные ассоциации, соединенные с одним классом,

Временные и пространственные ограничения обычно применяются при моделировании систем реального времени (см. главу 24).

символизируют различные связи. Благодаря ограничениям вы можете добавить новую семантику к существующим правилам. *Ограничения* (constraints)определяют условия, которым должна удовлетворять система времени исполнения, чтобы соответствовать модели. Например, вам надо показать, что взаимодействие по некоей ассоциации должно быть безопасным (рис. 6.8); конфигурации, нарушающие это ограничение, не соответствуют модели. Аналогичным образом вы можете указать, что определенный экземпляр может иметь связь только с одной из множества ассоциаций, соединенных с неким классом.

На заметку. Ограничения могут быть записаны в виде текста произвольного формата. Если вы хотите специфицировать вашу семантику более точно, можете использовать язык объектных ограничений UML (Object Constraint Language – OCL), подробно описанный в книге «UML» (ее выходные данные вы найдете во введении, раздел «Цели»).

Ограничения могут быть присоединены к нескольким элементам за счет использования зависимостей (см. главу 5).

Ограничения представлены строками, заключенными в фигурные скобки и помещенными рядом с ассоциированным элементом. Эта нотация применяется и для дополнения базовой нотации элемента с целью изображения спецификаций элемента, допускающих графическое представление. Например, некоторые свойства ассоциаций (порядок и изменчивость) изображаются с помощью ограничений.

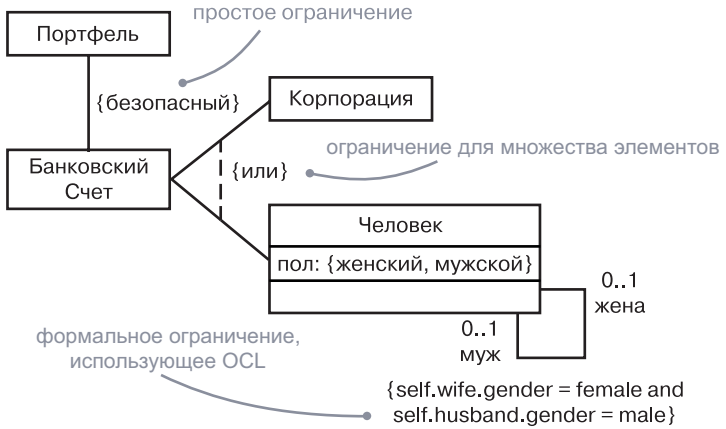


Рис. 6.8. Ограничения

Стандартные элементы

Классификаторы обсуждаются в главе 9.

UML определяет ряд стандартных стереотипов для классификаторов, компонентов, связей и других элементов моделирования. Существует один стандартный стереотип (представляющий интерес в основном для разработчиков инструментальных средств), который сам позволяет моделировать другие стереотипы:

- `stereotype` – определяет, что данный классификатор является стереотипом и может применяться к другим элементам.

Вы можете использовать данный стереотип, если нужно явно показать в модели те стереотипы, которые вы определили для своего проекта.

Профили

Зачастую бывает полезно адаптировать версию UML под конкретные цели или предметную область. Например, если вы хотите использовать модель UML для генерации кода на том или ином языке, вам может пригодиться определение стереотипов, которые, будучи применены к элементам, дадут подсказки генератору кода (подобно стереотипу `pragma` в языке Ада). Однако определяемые вами с этой целью стереотипы должны быть разными для Java и C++. Другой пример: вы моделируете с помощью UML базы данных. Некоторые функции языка, в частности динамическое моделирование, не слишком для вас важны, но при этом необходимо добавить новые концепции – скажем, возможные ключи и индексы. Вы можете адаптировать UML, используя профили.

Профиль (profile) – это UML-модель с набором готовых стереотипов, помеченных значений, ограничений и базовых классов. Также профиль включает подмножество типов элементов UML, используемых автором моделей для того, чтобы исключить те из них, которые не понадобятся в работе с определенной областью приложений. Иными словами, профили определяют специализированные версии UML, предназначенные для моделирования определенной предметной области. Поскольку профили построены на базе стандартного UML, они не определяют нового языка и могут быть поддержаны стандартными средствами UML.

Большинство разработчиков моделей не конструирует собственных профилей. В основном профили создаются разработчиками инструментальных средств, каркасов и т.п. Однако многие авторы моделей используют профили. В этом смысле они сродни библиотекам подпрограмм: немногочисленные эксперты создают их, чтобы затем ими воспользовались многие программисты (в частности, в целях поддержки языков программирования и баз данных, различных платформ, инструментов моделирования, а также разнообразных областей бизнес-приложений).

Типичные приемы моделирования

Моделирование комментариев

Общее назначение примечаний – запись комментариев, оценок или пояснений в свободной форме. Размещая эти комментарии непосредственно в модели, вы превращаете ее в «склад» разнообразных артефактов, создаваемых во время разработки. Вы даже можете использовать примечания для визуализации требований и демонстрации того, как они явно связаны с частями вашей модели.

При моделировании комментариев учитывайте следующее:

- Вводите комментарий в виде текста в примечание и размещайте его рядом с элементом, к которому он относится. Вы можете показать более явную связь примечания с элементом, установив между ними зависимость.
- Помните, что вы можете скрыть или показать элементы вашей модели там, где это понадобится. Вы не обязаны делать ваши комментарии видимыми, если видимы элементы, к которым они присоединены. Используйте комментарии на диаграммах только тогда, когда это важно для передачи существенной информации.
- Если комментарий слишком длинный или включает в себя нечто большее, че м простой текст, рассмотрите возможность его размещения во внешнем документе и встраивания последнего в примечание, присоединенное к вашей модели (либо связывания этого документа с примечанием).
- По мере развития вашей модели сохраняйте комментарии, отражающие только существенные решения, которые невозможно извлечь из самой модели. Остальные же удаляйте, если только они не представляют исторического интереса.

Простое обобщение обсуждается в главе 5; более сложные формы обобщения описаны в главе 10.

Например, рис. 6.9 демонстрирует модель, которая находится в процессе разработки иерархии классов, показывая некоторые требования, формирующие модель, а также некоторые примечания к ней, высказанные при оценке дизайна.

В этом примере большинство комментариев представлены в виде простого текста (такого, как замечание Мэри), но один из них (примечание в нижней части диаграммы) представляет собой гиперссылку на другой документ.

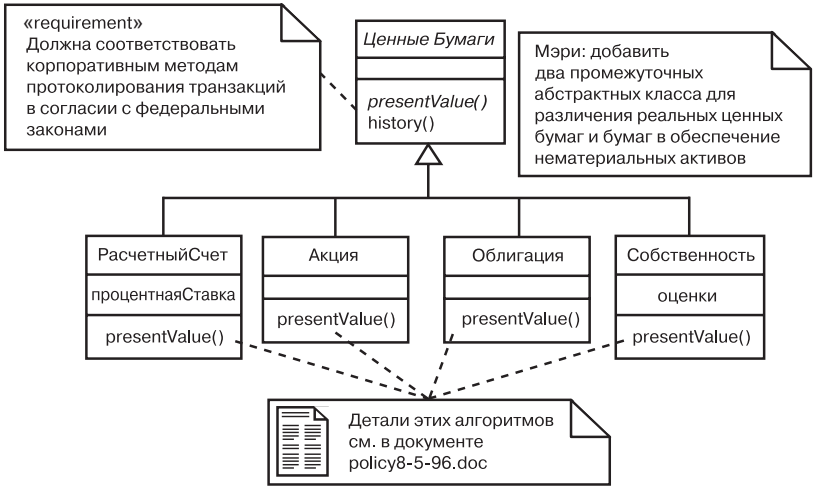


Рис. 6.9. Моделирование комментариев

Моделирование новых сойств

Базовые свойства строительных блоков UML – атрибуты и операции для классов, содержимое пакетов и др. – носят достаточно универсальный характер для выражения большинства моделируемых сущностей. Однако если вы хотите расширить свойства этих базовых строительных блоков, вам придется определять стереотипы и помеченные значения.

Чтобы смоделировать новые свойства, необходимо:

- Убедиться в том, что не существует способа решить поставленную вами задачу средствами стандартного UML.
- Определить стереотип и добавить к нему новые свойства. Здесь применимы правила обобщения: помеченные значения, определенные для некоторого вида стереотипов, применимы также к его потомкам.

Подсистемы обсуждаются в главе 32.

Предположим, что вы хотите привязать созданные вами модели к системе управления конфигурацией вашего проекта. Помимо всего прочего, это означает необходимость отслеживать номера версий, текущий статус «check in/check out»[3] и, может быть, даже дату и время создания/модификации каждой из подсистем. Поскольку все это специфичная для процесса информация, она не является базовой частью UML (хотя вы можете добавить ее в виде помеченных значений).

[3] Имеется в виду состояние рабочей копии на машине разработчика по отношению к репозиторию проекта. – *Прим. перев.*

Более того, эта информация не выражается атрибутами классов. Номер версии подсистемы – часть метаданных, а не часть модели.

Рис. 6.10 демонстрирует три подсистемы, каждая из которых расширена стереотипом «versioned» для указания номера версии и статуса.

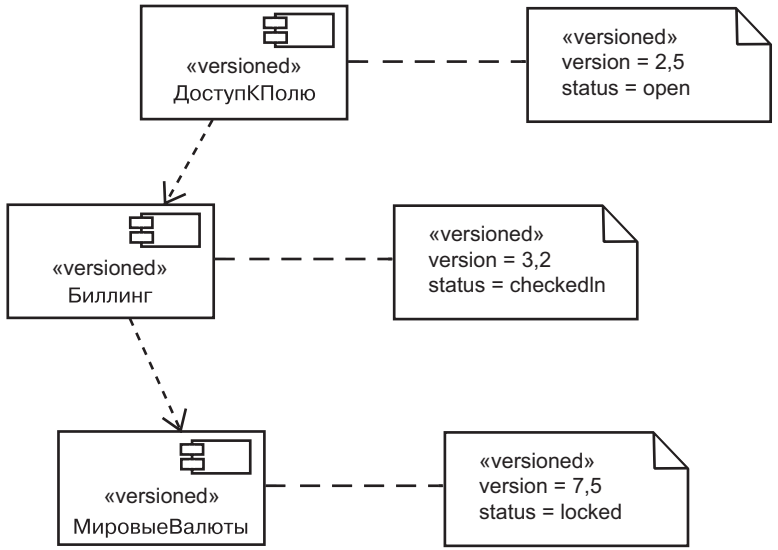


Рис. 6.10. Моделирование новых свойств

На заметку. Значения тегов, такие как `version` (версия) или `status` (состояние), могут быть установлены в ваших моделях инструментальными средствами. Вместо того, чтобы устанавливать эти величины вручную, вы можете использовать среды разработки, интегрированные с инструментами управления конфигурацией и инструментами моделирования.

Моделирование новой семантики

Создавая модель с помощью UML, вы работаете по определенным правилам данного языка. И не напрасно: благодаря этому вы можете четко передать свои намерения любому, кто умеет читать UML-модели. Но если вам требуется выразить новую семантику, о которой UML умалчивает или же модифицировать правила UML, придется писать ограничение.

Чтобы смоделировать новую семантику, воспользуйтесь следующими рекомендациями:

- Убедитесь в том, что не существует способа решить поставленную вами задачу средствами стандартного UML.
- Опишите новую семантику в ограничении, помещенном рядом с элементом, к которому она относится. Вы можете явно продемонстрировать эту связь, установив зависимость между ограничением и элементом.
- Если вам нужно более точно и формально специфицировать семантику, опишите ее на языке OCL.

В качестве примера на рис. 6.11 представлена модель небольшой части корпоративной системы управления человеческими ресурсами.

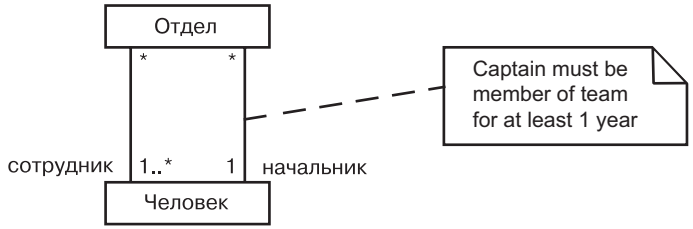


Рис. 6.11. Моделирование новой семантики

Диаграмма показывает, что каждый человек (`Person`) может быть членом одной или нескольких команд (`Team`) либо не входить ни в одну из них; в каждой команде должен быть как минимум один руководитель; каждый человек может быть руководителем одной или нескольких команд либо не руководить ни одной. Вся эта семантика может быть выражена простыми средствами UML. Однако тот факт, что руководитель должен быть членом команды, которую он возглавляет, касается нескольких ассоциаций и не может быть выражен с использованием стандартного UML. Чтобы задать этот инвариант, вы должны описать ограничение, которое показывает, что руководитель относится к числу членов команды, и связать обе ассоциации ограничением. Еще одно ограничение указывает, что руководитель должен быть членом команды не меньше года.

Советы и подсказки

Снабжая модель примечаниями, пользуйтесь следующими правилами:

- добавляйте только те требования, оценки и пояснения, которые вы не можете выразить просто и четко другими средствами UML;
- используйте примечания как своеобразные электронные памятки для отслеживания хода работы;

- ❑ не перегружайте модель большими блоками комментариев. Если вам нужны обстоятельные комментарии, используйте примечания, которые указывают на документы, содержащие полный текст пояснений.

При расширении модели стереотипами, помеченными значениями или ограничениями:

- ❑ стандартизируйте небольшой набор стереотипов, помеченных значений и ограничений, предназначенных для использования в вашем проекте, и не позволяйте отдельным разработчикам создавать новые расширения;
- ❑ задавайте короткие информативные имена стереотипов и помеченных значений;
- ❑ если требования точности не слишком строги, используйте для спецификации ограничения текст произвольного формата. В противном случае применяйте OCL для описания ограничений.

При изображении стереотипа, помеченного значения или ограничения:

- ❑ используйте графические стереотипы умеренно. Вы можете полностью переопределить базовую нотацию UML посредством стереотипов, но тем самым вы затрудните понимание моделей посторонними пользователями;
- ❑ помните о возможности цветового выделения графических стереотипов и сложных пиктограмм. Как правило, простая нотация всегда лучше – решение модели в цвете может облегчить ее понимание.

Глава 7. Диаграммы

В этой главе:

Общие принципы моделирования обсуждаются в главе 1.

Занимаясь моделированием, вы допускаете некоторое упрощение реальности с тем, чтобы лучше описать разрабатываемую систему. С помощью UML вы строите модели из базовых блоков: классов, интерфейсов, кооперaций, компонентов, узлов, зависимостей, обобщений и ассоциаций. Диаграммы изображают эти строительные блоки, помогая вам наглядно представить модель.

Диаграмма – это графическое представление набора элементов, чаще всего отображаемых как связный граф вершин (сущностей) и дуг (связей). Диаграммы используются для визуализации системы с разных точек зрения. Поскольку ни одна сложная система не может быть понята при одностороннем рассмотрении, UML определяет множество диаграмм, с тем чтобы вы могли сфокусировать внимание на различных аспектах системы, рассматриваемых обособленно.

Хорошие диаграммы облегчают понимание разрабатываемой системы. Выбор правильного множества диаграмм для моделирования вашей системы заставляет вас задавать правильные вопросы о системе и помогает высветить последствия принятых решений.

Введение

Работая с архитектором над дизайном дома, который вы собрались построить, вы для начала совершаете три действия: составляете список пожеланий (например, «В доме должно быть три ванных комнаты», «Я хочу уложиться в такую-то сумму» и т.д.), делаете ряд простых набросков, представляющих ключевые характеристики дома (допустим, набросок входа с винтовой лестницей и т.д.), а также некоторые общие идеи, касающиеся стиля (скажем, ««французский сельский домик»). Работа архитектора состоит в том, чтобы все эти отрывочные, изменчивые и возможно, противоречивые пожелания воплотить в дизайне.

Вероятно, начнет он с плана первого этажа. Этот артефакт ляжет в основу общего представления о доме, после чего уже можно будет уточнить детали и документировать принятые решения. При каждом просмотре вы захотите внести некоторые изменения – перепроектировать комнаты, по-другому разместив стены, двери и окна. На начальной стадии чертежи меняются часто, но по мере проработки дизайна он все больше соответствует вашим требованиям к форме и функциям тех или иных объектов, рабочему времени и затратам; наконец, чертежи стабилизируются настолько, что можно утвердить их и начать строительство. Но даже после этого не исключено, что вы измените ряд диаграмм и создадите какие-то новые.

Разумеется, помимо плана первого этажа вам понадобятся другие представления дома. Например, вы захотите увидеть его с разных сторон. Кроме того, архитектору понадобится составить планы электропроводки, системы отопления и вентиляции, а также водопровода и канализации. Если дизайн предполагает нестандартные решения (скажем, использование арочных пролетов или размещение домашнего кинотеатра в непосредственной близости от камина), придется делать дополнительные наброски.

Практика создания диаграмм для представления систем с разных точек зрения широко распространена: без нее не обойтись в любой инженерной дисциплине, имеющей дело со сложными системами, – от строительства жилых домов до авиа- и судостроения, а также разработки программного обеспечения.

Пять представлений архитектуры обсуждаются в главе 2.

Как уже говорилось выше, существуют пять взаимодополняющих представлений, которые особенно важны для визуализации, специфицирования, конструирования и документирования программной архитектуры: представление вариантов использования, представление дизайна, представление взаимодействия, представление реализации и представление развертывания. Каждое из этих представлений включает структурное моделирование (моделирование статических

сущностей), а также поведенческое моделирование (моделирование динамических сущностей). Все эти разнообразные представления в своей совокупности охватывают наиболее важные решения, касающиеся системы; каждое в отдельности позволяет вам сфокусировать внимание на одном ее аспекте таким образом, чтобы вы могли четко оценить принятые проектные решения.

Моделирование архитектуры системы обсуждается в главе 32.

Рассматривая программную систему с любой точки зрения с помощью UML, вы используете диаграммы для организации интересующих вас элементов. Разные виды диаграмм можно комбинировать для создания того или иного представления. Например, статические аспекты представления реализации системы могут быть визуализированы на диаграмме классов, а динамические аспекты того же представления – на диаграмме взаимодействий.

Конечно, вы не ограничены предопределенными типами диаграмм. Существующие в UML типы диаграмм представляют часто используемые способы объединения просматриваемых элементов, но не более того. Чтобы учесть нужды вашего предприятия, вы вправе создать свои собственные виды диаграмм.

Пошаговый итерационный процесс рассматривается в приложении 2.

Диаграммы UML можно использовать двумя основными способами: для специфицирования моделей, на основе которых вы конструируете исполняемую систему (прямое проектирование), и для реконструкции моделей на основе частей существующих исполняемых систем (обратное проектирование). В каждом случае, подобно тому как это происходит при строительстве дома, вам придется выстраивать диаграммы шаг за шагом, добавляя по одному элементу, и итерационно, повторяя цикл «немножко проектируем – немножко строим».

Базовые понятия

Системы, модели и представления описываются в главе 32.

Система – это набор подсистем, организованных для достижения определенной цели и описанных с помощью набора моделей (возможно, с различных точек зрения).

Подсистема – группа элементов, часть которых составляет спецификацию поведения, представленного другими ее составляющими.

Модель – семантически завершенная абстракция системы, которая создана по принципу полного и самодостаточного упрощения реальности, ставящего целью лучшее понимание системы.

В контексте архитектуры *представление* – это проекция организации и структуры системной модели, сфокусированная на одном из ее аспектов.

Диаграмма – графическое представление набора элементов, чаще всего изображаемых в виде связного графа вершин (сущностей) и дуг (связей).

Чтобы увязать все вышеперечисленное, выразимся так: система – это ваша разработка, представленная с разных точек зрения разными моделямис использованием диаграмм.

Проще говоря, диаграмма – графическая проекция элементов, составляющих систему. Например, в проекте корпоративной системы управления человеческими ресурсами может быть несколько сотен классов. Вы никогда не визуализируете структуру или поведение системы на одной огромной диаграмме, включающей в себя все эти классы и их связи. Вместо этого потребуется создать несколько диаграмм, каждая из которых будет сфокусирована на одном представлении. Например, диаграмма классов, включающая такие элементы, как `Person` (Человек), `Department` (Отдел) и `Office` (Офис), отражает схему базы данных. Некоторые из этих классов вы наряду с другими выведете на диаграмме, представляющей API – программный интерфейс, используемый клиентскими приложениями). Весьма вероятно, что те же классы частично появятся на диаграмме взаимодействий, специфицирующей семантику транзакции, которая переназначает сотрудника (`Person`) в новый отдел (`Department`).

Как показывает этот пример, одна и та же сущность в системе – например, класс `Person`, – может неоднократно встречаться на одной и той же диаграмме или в разных диаграммах. При этом каждая диаграмма обеспечивает свое представление элементов, из которых состоит система.

При моделировании реальных систем (неважно, из какой проблемной области) вы обнаруживаете, что приходится создавать одни и те же виды диаграмм, поскольку они представляют общие взгляды на общие модели. Как правило, статические части системы описываются одной из следующих диаграмм:

1. Диаграмма классов;
2. Диаграмма компонентов;
3. Диаграмма составной структуры;
4. Диаграмма объектов;
5. Диаграмма размещения;
6. Диаграмма артефактов.

Часто вы также будете использовать пять дополнительных диаграмм для представления динамических частей систем:

1. Диаграмма вариантов использования;
2. Диаграмма последовательности;
3. Диаграмма коммуникации;
4. Диаграмма состояний;
5. Диаграмма деятельности.

Пакеты обсуждаются в главе 12.

Каждая диаграмма, которую вы создаете, скорее всего будет относиться к одному из этих девяти типов или же к какому-либо другому, определенному специально для вашего проекта или организации.

Каждой диаграмме должно быть назначено имя, уникальное в своем контексте, чтобы вы могли обратиться к ней и отличить от других. При проектировании большинства систем, кроме самых тривиальных, диаграммы желательно объединить в пакеты.

Вы можете представить на одной диаграмме любую комбинацию элементов UML. Например, на одной и той же диаграмме можно вывести классы и объекты (это распространенный прием) или классы и компоненты (допускается, но реже). Хотя ничто не мешает вам разместить несопоставимые элементы моделирования на одной диаграмме, все же предпочтительнее обходиться подобными видами сущностей.

Типы диаграмм UML фактически определяются элементами, которые чаще всего на них изображаются. Так, если вы хотите визуализировать набор классов и их связей, то используете диаграмму классов; если представляете набор компонентов, – диаграмму компонентов и т.д.

Структурные диаграммы

Структурные диаграммы UML предназначены для визуализации, специфицирования, конструирования и документирования статических аспектов системы. Последние можно образно представить как относительно прочный «скелет» системы или «строительные леса». Подобно тому как статические аспекты дома включают в себя наличие и местоположение стен, дверей, окон, труб, проводов, вентиляции, так и статические аспекты программной системы включают в себя наличие и размещение классов, интерфейсов, кооперации, компонентов и узлов.

Структурные диаграммы UML преимущественно включают в себя основные группы сущностей, которые вы используете при моделировании системы:

1. Диаграмма классов	классы, интерфейсы и кооперации
2. Диаграмма компонентов	компоненты
3. Диаграмма составной структуры	внутренняя структура
4. Диаграмма объектов	объекты
5. Диаграмма размещения	узлы
6. Диаграмма артефактов	артефакты

Диаграмма классов показывает набор классов, интерфейсов и кооперации, а также их связи. Чаще других диаграмм используется при моделировании объектно-ориентированных систем. Иллюстрирует статическое представление дизайна системы. Диаграмма классов, которая включает активные классы, определяет статический взгляд на процессы системы.

Диаграммы составной структуры и диаграммы компонентов обсуждаются в главе 15.

Диаграмма компонентов показывает внутренние части, коннекторы и порты, реализующие компонент. При создании экземпляра компонента создаются и экземпляры его внутренних частей.

Диаграмма составной структуры представляет внутреннюю структуру класса или кооперации. Разница между диаграммой компонентов и диаграммой составной структуры невелика, поэтому в этой книге они трактуются как диаграммы компонентов.

Диаграммы объектов обсуждаются в главе 14.

Диаграмма объектов показывает набор объектов и их связи. Используется для описания структур данных, статических снимков экземпляров сущностей, выведенных на диаграмме классов. Служит для статического представления дизайна или процессов системы, как и диаграмма классов, но, в отличие от последней, – с точки зрения реальных или прототипных ситуаций.

Диаграммы артефактов обсуждаются в главе 30.

Диаграмма артефактов показывает набор артефактов и их связей с другими артефактами, а также классами, которые они реализуют. Используется для представления физической реализации элементов системы. UML рассматривает диаграммы артефактов как разновидность диаграмм размещения, но мы выделяем их для удобства описания.

Диаграммы размещения обсуждаются в главе 31.

Диаграмма размещения показывает набор узлов и их связей. Описывает статическое представление архитектуры. Связана с диаграммой компонентов (на каждом узле обычно размещается один или несколько компонентов).

На заметку. Существуют некоторые общие разновидности четырех типов диаграмм, перечисленных выше. Они именуются в соответствии с их главным назначением. Например, вы можете создать диаграмму подсистем, чтобы проиллюстрировать структурную декомпозицию системы на ряд подсистем. По сути, это обычная диаграмма классов, которая в основном содержит подсистемы.

Поведенческие диаграммы

Поведенческие диаграммы UML используются для визуализации, специфицирования, конструирования и документирования динамических аспектов системы, то есть для представления ее изменяющихся частей. Если динамические аспекты дома предусматривают перемещение потоков воздуха и людей по комнатам, то динамические аспекты программной системы учитывают поток сообщений во времени и физическое перемещение компонентов по сети.

Поведенческие диаграммы UML позволяют моделировать динамику системы в основном следующими способами:

1. Диаграмма вариантов использования организует поведение системы;
2. Диаграмма последовательности сфокусирована на временной последовательности сообщений;
3. Диаграмма коммуникации сфокусирована на структурной организации объектов, отправляющих и принимающих сообщения;
4. Диаграмма состояний сфокусирована на изменении состояния системы в ответ на события;
5. Диаграмма деятельности сфокусирована на потоке управления от деятельности к деятельности.

Диаграммы вариантов использования обсуждаются в главе 18.

Диаграмма вариантов использования показывает набор вариантов использования и действующих лиц (как особую разновидность классов), а также их связи. Иллюстрирует статическое представление вариантов использования системы. Особенно важна для организации и моделирования поведения системы.

Диаграмма взаимодействия – это общее название диаграмм последовательности и коммуникации. Все диаграммы последовательности и коммуникации являются диаграммами взаимодействия, а все диаграммы взаимодействия являются либо диаграммами последовательности, либо диаграммами коммуникации. В их основе одна и та же модель, хотя на практике они включают разные сущности. Еще один вид диаграмм взаимодействия – временные диаграммы, но в данной книге мы их рассматривать не будем.

Диаграммы последовательности обсуждаются в главе 19.

Диаграмма последовательности – это диаграмма взаимодействия, куда входят упорядоченные по времени сообщения, отправляемые и принимаемые экземплярами, которые играют роли. Иллюстрирует динамическое представление системы.

Диаграммы коммуникации обсуждаются в главе 19.

Диаграмма коммуникации – это диаграмма взаимодействия, показывающая структурную организацию объектов, отправляющих и принимающих сообщения. Изображает набор ролей, коннекторов между ними, а также сообщения, отправляемые и принимаемые экземплярами, которые играют эти роли. Иллюстрирует динамическое представление системы.

Диаграммы состояний обсуждаются в главе 25.

Диаграмма состояний показывает конечный автомат (state machine), состоящий из набора состояний, переходов, событий и деятельностей. Иллюстрирует динамическое представление системы. Диаграммы этого типа особенно важны при моделировании поведения интерфейса, класса или кооперации. На них ясно прослеживается упорядоченное по событиям поведение объекта, что особенно удобно при моделировании реактивных систем.

Диаграммы деятельности обсуждаются в главе 19.

Диаграмма деятельности показывает шаг за шагом поток вычислений. Деятельность описывает набор действий, последовательный или ветвящийся их поток и значения, генерируемые или используемые действиями. Диаграммы деятельности иллюстрируют динамическое

представление системы. Они особенно важны при моделировании функций системы. Акцентируют внимание на потоке управления при реализации некоторого поведения.

На заметку. Существуют очевидные практические ограничения при иллюстрировании динамических сущностей (поведения системы) с помощью диаграмм, которые, по сути своей, являются статическими артефактами, особенно когда вы рисуете их на бумаге, доске и т.д. Изображенные на экране монитора, поведенческие диаграммы могут быть анимированы, с тем чтобы имитировать работающую систему либо передать реальное ее поведение. UML позволяет создавать динамические диаграммы и применять цвета и другие визуальные средства для «запуска» диаграмм (то есть более наглядного представления динамики). Некоторые инструментальные средства уже демонстрируют такое «продвинутое» использование UML.

Типичные приемы моделирования

Моделирование различных представлений системы

При моделировании системы с разных точек зрения вы, по сути, параллельно конструируете ее во многих измерениях. Выбирая правильный набор представлений, вы запускаете процесс, который помогает вам ставить правильные вопросы о системе и выявлять риски, которым она будет подвергаться. Если вынеграмотно подойдете к выбору этих представлений, например сосредоточитесь преимущественно на одном из них в ущерб всем прочим, некоторые важные обстоятельства могут выпасть из вашего поля зрения – таким образом, в будущем вы столкнетесь с проблемами, которые серьезно повредят работе над проектом.

Чтобы смоделировать систему с разных точек зрения, необходимо:

- Решить, какие представления понадобятся для наилучшего выражения архитектуры вашей системы и для снижения технических рисков проекта. Пять вышеописанных представлений архитектуры – базовая информация для начала такой работы.
- Определить, какие артефакты вам понадобятся для того, чтобы охватить наиболее существенные детали каждого выбранного представления. Чаще всего в качестве таких артефактов выступают разнообразные диаграммы UML.
- Отметить диаграммы, для которых следует предусмотреть формальный или полуформальный контроль (то есть те, которые требуют периодического пересмотра), и сохранить их в составе документации проекта.
- Оставить место для диаграмм, которые пока не задействованы. Эти временные диаграммы помогут исследовать реализацию ваших решений и продумать возможные изменения моделируемой системы.

К примеру, вы моделируете простое монолитное приложение, которое исполняется на одном компьютере. Для этого потребуется не так уж много диаграмм:

- представление вариантов использования – диаграммы вариантов использования
- представление дизайна – диаграммы классов (для структурного моделирования)
- представление взаимодействий – диаграммы взаимодействия (для поведенческого моделирования)
- представление реализации – диаграммы составной структуры
- представление развертывания – –

Если ваша система реактивная или же сосредоточена на потоке процессов, не исключено, что для моделирования ее поведения вы захотите использовать соответственно диаграммы состояний и диаграммы деятельности.

Если же вы работаете над системой с архитектурой «клиент/сервер», вам, вероятно, понадобятся диаграммы компонентов и диаграммы развертывания, чтобы смоделировать ее физические особенности.

Наконец, при подготовке сложной распределенной системы вам придется задействовать полный набор диаграмм UML, чтобы выразить ее архитектуру и выявить технические риски проекта:

- представление вариантов использования – диаграммы вариантов использования, диаграммы последовательности
- представление дизайна – диаграммы классов (для структурного моделирования)

	диаграммы взаимодействия (для поведенческого моделирования) диаграммы состояний для поведенческого моделирования) диаграммы деятельности
❑ представление взаимодействий	диаграммы взаимодействия (для поведенческого моделирования)
❑ представление реализации	диаграммы классов диаграммы составной структуры
❑ представление развертывания	диаграммы развертывания

Моделирование различных уровней абстракции

Не только вам понадобится оценивать систему с разных точек зрения – некоторые пользователи нуждаются в тех же представлениях системы, но на разных уровнях абстракции. Например, имея набор классов, который охватывает словарь предметной области, программист может пожелать уточнить представление вплоть до уровня атрибутов, операций и связей. В то же время аналитик, исследующий сценарии вариантов использования вместе с конечным пользователем, пожелает видеть более общее представление тех же классов. В этом контексте программист работает на более низком уровне абстракции, а аналитик и конечный пользователь – на более высоком, хотя все они имеют дело с одной и той же моделью. Поскольку диаграммы – это всего лишь графическое представление элементов, составляющих модель, вы можете создавать несколько диаграмм на одной модели либо несколько моделей с разной степенью детализации, причем в каждом случае скрываются или выводятся различные наборы элементов.

Есть два основных пути моделирования систем на разных уровнях абстракции: либо создаются диаграммы одной и той же модели с разной степенью детализации, либо формируются модели на разных уровнях абстракции с диаграммами, отражающими переход от одной модели к другой.

Чтобы смоделировать систему с разными уровнями детализации, необходимо:

- ❑ Рассмотреть потребности будущих пользователей и начать с заранее выбранной модели.

- ❑ Если вашу модель будут использовать для конструирования реализации, следует показать на диаграммах значительное число деталей на диаграммах (то есть их уровень абстракции относительно низкий). Если же с помощью данной модели планируется представить концепцию системы конечным пользователям, вам понадобятся диаграммы, находящиеся на более высоком уровне абстракции – это означает скрытие большинства деталей.
- ❑ Создать диаграммы с выбранным уровнем абстракции, скрывая или отображая четыре категории сущностей модели:

Сообщения обсуждаются в главе 16; переходы – в главе 22; стереотипы – в главе 6.

1. *Строительные блоки и связи:* скройте те из них, которые не отражают назначение вашей диаграммы и не отвечают запросам пользователей;
2. *Дополнения:* показывайте их только для тех строительных блоков и связей, которые важны для понимания ваших намерений;
3. *Поток:* на поведенческих диаграммах раскрывайте только те сообщения и переходы, которые проясняют ваши намерения;
4. *Стереотипы*, используемые для классификации списков сущностей (таких как атрибуты и операции): показывайте только те элементы со стереотипами, которые раскрывают ваши намерения.

Главное преимущество такого подхода в том, что вы всегда моделируете на основе общего семантического репозитория. Главный же недостаток в том, что изменения в диаграмме одного уровня абстракции могут сделать недействительными диаграммы других уровней абстракции.

Для моделирования системы на разных уровнях абстракции с созданием разноуровневых моделей необходимо:

- ❑ С учетом потребностей пользователей определить уровень абстракции для каждого представления и сформировать отдельную модель для каждого уровня.
- ❑ Наполнить модели, находящиеся на высоком уровне абстракции, простыми абстракциями, а модели, находящиеся на низком уровне абстракции – детализированными. Установить трассировку зависимостей между связанными элементами разных моделей.
- ❑ Если вы следуете пяти представлениям архитектуры, при моделировании систем на разных уровнях абстракции возможны четыре общих ситуации:

1. Варианты использования в модели вариантов использования будут связаны с кооперациями в модели дизайна;

2. Кооперации будут связаны с совокупностью классов, работающих вместе в их составе;
3. Компоненты в модели реализации будут связаны с элементами модели дизайна;
4. Узлы в модели развертывания будут связаны с компонентами в модели реализации.

Варианты использования обсуждаются в главе 17; кооперации – в главе 28; компоненты – в главе 15; узлы – в главе 27.

Главное преимущество данного подхода в том, что диаграммы разных уровней абстракции остаются слабо связанными. Это значит, что изменения в одной модели не окажут сильного влияния на другие модели. Главный недостаток: потребуется немало усилий на то, чтобы синхронизировать все модели и их диаграммы. Это особенно важно, когда ваши модели относятся к разным фазам жизненного цикла разработки программного обеспечения (например, если вы решите сопровождать аналитическую модель отдельно от модели дизайна).

Диаграммы взаимодействия обсуждаются в главе 19.

Предположим, что вы моделируете систему для Web-коммерции. Одним из основных вариантов ее использования будет размещение заказа. Если вы аналитик или конечный пользователь, то вероятно, создадите некоторые диаграммы взаимодействия на высоком уровне абстракции, описывающие действие по размещению заказа так, как показано на рис. 7.1.

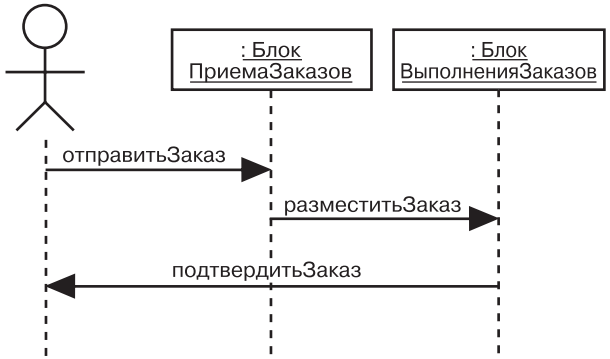


Рис. 7.1. Диаграмма взаимодействия на высоком уровне абстракции

Между тем программист, отвечающий за реализацию данного сценария, должен построить другую диаграмму, где приводятся точные сообщения и добавляются новые игроки в процесс взаимодействия (рис. 7.2).

Обе диаграммы обрисовывают одну и ту же модель, но с разной степенью детализации. Вторая диаграмма содержит дополнительные сообщения и роли. Целесообразно подготовить много диаграмм вроде этой, особенно если ваш инструментарий облегчает навигацию от одной диаграммы к другой.

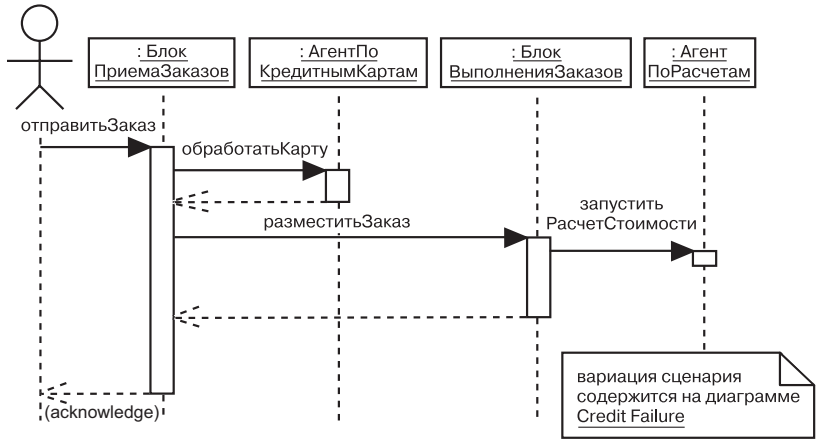


Рис. 7.2. Диаграмма взаимодействия на низком уровне абстракции

Моделирование сложных представлений

Независимо от того, как вы разобьете свою модель, рано или поздно вы столкнетесь с необходимостью создания больших сложных диаграмм. Например, анализируя полную схему базы данных, включающую 100 или более абстракций, полезно будет изучить диаграмму, показывающую *все* эти классы и их ассоциации. Благодаря этому вы сможете увидеть некоторые общие образцы кооперации. Если же вы рассмотрите модель на более высоком уровне абстракции, скрывающей детали, то потеряете информацию, необходимую для понимания.

При моделировании сложного представления необходимо:

Пакеты обсуждаются в главе 12; кооперации – в главе 28.

- Убедиться, что не существует способа представить ту же информацию на более высоком уровне абстракции (возможно, скрывая некоторые части диаграммы и оставляя другие части высокодетализированными).
- Если вы скрыли максимум деталей, но диаграмма все еще сложна для восприятия, – рассмотреть возможность группировки некоторых элементов в пакеты или более высокоуровневые кооперации, а затем изменить диаграмму, представив на ней только эти пакеты или кооперации.
- Если диаграмма по-прежнему остается сложной, – использовать примечания и цвета, чтобы сконцентрировать внимание читателей на элементах, требующих пояснения.
- Наконец, если и после этого диаграмма нуждается в корректировке, – распечатать ее полностью и вывесить на стене. При этом диаграмма теряет интерактивность, зато вы сможете сделать шаг назад и исследовать ее на предмет наличия общих образцов.

Советы и подсказки

При создании диаграмм:

- помните, что назначение диаграмм UML не в том, чтобы пользователь увидел симпатичные картинки, а в том, чтобы визуализировать, специфицировать, конструировать и документировать. Диаграммы – это одно из средств подготовки работающей системы;
- не все диаграммы стоит сохранять. Рассмотрите возможность построения диаграмм «на лету», анализируя элементы ваших моделей, и используйте их разумным образом в процессе построения системы. Многие диаграммы можно удалить после того, как они выполнят свое предназначение (но семантика, на основе которой они были построены, останется частью модели);
- не перегружайте модели излишними или избыточными диаграммами;
- показывайте на каждой диаграмме лишь минимум деталей, необходимый для того, чтобы она выполнила свое предназначение. Второстепенная информация может отвлечь пользователя от основной идеи, которую вы пытаетесь выразить;
- в то же время не создавайте чересчур упрощенных диаграмм, если только вам не нужно изобразить что-либо на очень высоком уровне абстракции. Чрезмерное упрощение может скрыть детали, важные для понимания ваших моделей;
- сохраняйте баланс между структурными и поведенческими диаграммами в вашей системе. Очень немногие системы являются полностью статическими или полностью динамическими;
- не делайте диаграммы слишком большими (те из них, что не умещаются на стандартном листе бумаги, трудно читать) или слишком маленькими (рассмотрите возможность объединения нескольких простых диаграмм в одну);
- присваивайте каждой диаграмме имя, которое четко поясняет ее назначение;
- организуйте ваши диаграммы – группируйте их в пакеты в соответствии с представлениями;
- не увлекайтесь форматированием диаграмм. Предоставьте это инструментальным средствам.

Хорошо структурированная диаграмма:

- сосредоточена на передаче одного аспекта системы;
- содержит только те элементы, которые существенны для понимания этого аспекта;

- детализирована в соответствии с уровнем абстракции (показывает только те дополнения, которые важны для понимания диаграммы на данном уровне);
- не настолько лаконична, чтобы пользователь упустил из виду важную семантику.

Когда вы рисуете диаграмму:

- расположите ее элементы так, чтобы свести к минимуму пересечение линий;
- стремитесь к тому, чтобы семантически близкие элементы располагались рядом;
- используйте примечания и выделения цветом, чтобы акцентировать важные детали диаграмм. Однако старайтесь не использовать похожие оттенки и применяйте цвета только с целью выделения, а не для передачи важной информации.

Глава 8. Диаграммы классов

В этой главе:

- Моделирование простых кооперацій
- Моделирование логической схемы базы данных
- Прямое и обратное проектирование

Диаграммы классов – это наиболее часто используемый тип диаграмм, которые создаются при моделировании объектно-ориентированных систем. Показывают набор классов, интерфейсов и кооперацій, а также их связи.

На практике диаграммы классов применяют для моделирования статического представления системы (по большей части это моделирование словаря системы, кооперацій или схем). Кроме того, диаграммы данного типа служат основой для целой группы взаимосвязанных диаграмм – диаграмм компонентов и диаграмм размещения.

Диаграммы классов важны не только для визуализации, специфицирования и документирования структурных моделей, но также для конструирования исполняемых систем посредством прямого и обратного проектирования.

Введение

В любом деле, например строительстве дома, не обойтись без понятийного аппарата – в данном случае словаря, который включает основные строительные блоки: стены, полы, окна, двери, потолки, перекрытия. Эти элементы в значительной мере структурированы (стены имеют высоту, ширину и толщину), но притом каждый из них ведет себя определенным образом (стены разных типов несут разные нагрузки; двери могут открываться и закрываться в разные стороны; существуют ограничения на размер пролетов, образуемых перекрытиями). Вы не можете рассматривать эти структурные и поведенческие параметры независимо друг от друга: при строительстве понадобится учитывать, как они *взаимодействуют*. Процесс проектирования вашего дома, таким образом, включает в себя компоновку всех вышеперечисленных сущностей в уникальной манере, подчиняющейся велениям здравого смысла и призванной удовлетворить все ваши функциональные и нефункциональные требования.

Разработка программного обеспечения ведется примерно по той же схеме – за исключением того, что благодаря гибкости ПО вы имеете возможность определять «с нуля» свои собственные базовые строительные блоки. Диаграммы классов в UML используются для того, чтобы визуализировать статические аспекты этих строительных блоков и их связей, а также обозначить детали их конструирования (см. рис. 8.1).

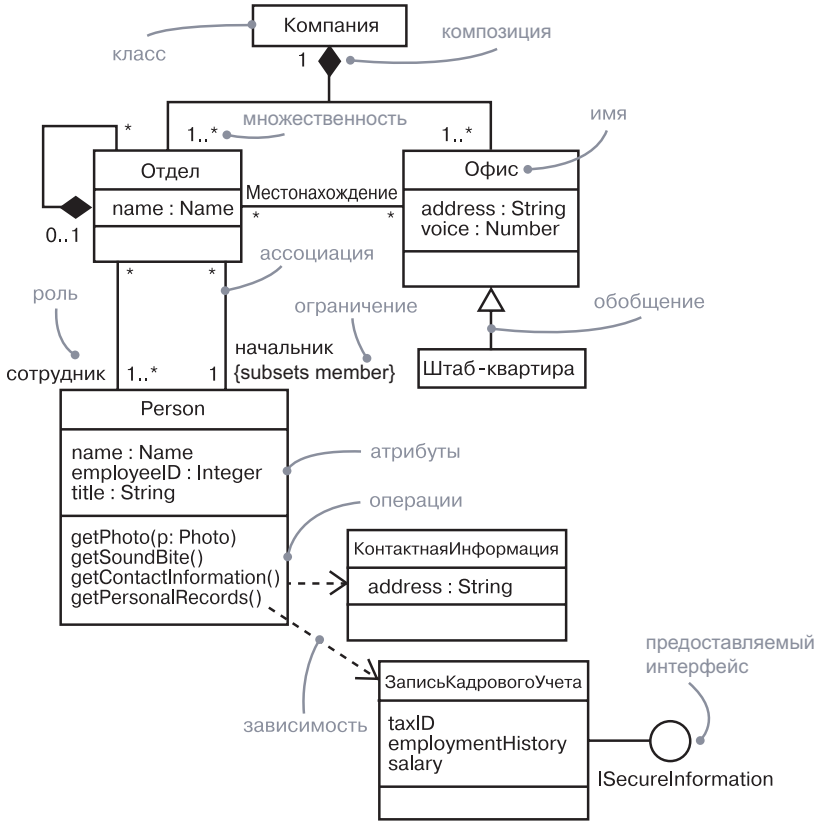


Рис. 8.1. Диаграмма классов

Базовые понятия

Диаграмма классов – это диаграмма, которая показывает набор классов, интерфейсов, кооперации и их связи. Графически представляет собой набор вершин и связывающих их дуг.

Общие свойства

Общие свойства диаграмм обсуждаются в главе 2.

Диаграмма классов, как и любая другая диаграмма, обладает именем и содержимым, которое является проекцией модели. От других типов диаграмм отличается конкретным наполнением.

Содержимое

Классы обсуждаются в главах 4 и 9; интерфейсы – в главе 11; связи – в главах 5 и 10; пакеты – в главе 12; подсистемы – в главе 32; экземпляры – в главе 13.

На диаграммах классов обычно представлены следующие элементы:

- классы;
- интерфейсы;
- зависимости, обобщения и ассоциации.

Как и другие диаграммы, они могут содержать примечания и ограничения.

Также могут включать в себя пакеты или подсистемы; те и другие группируют элементы модели в более крупные образования. Иногда в диаграмму классов требуется добавить экземпляры, особенно если вы хотите визуализировать тип экземпляра (возможно, динамический).

На заметку. Диаграммы компонентов и диаграммы размещения похожи на диаграммы классов, за исключением того что вместо классов они содержат соответственно компоненты и узлы.

Стандартное использование

Представления дизайна обсуждаются в главе 2.

Вы используете диаграммы классов для моделирования статического представления системы. Это представление, прежде всего, поддерживает ее функциональные требования, то есть описывает услуги, которые система должна предоставлять ее конечным пользователям.

Моделирование словаря системы обсуждается в главе 4.

Прорабатывая статическое представление системы, вы обычно используете диаграммы классов с одной из трех целей:

1. Для моделирования словаря системы.

Моделирование словаря системы предполагает решение вопроса о том, какие абстракции станут предметом внимания системы, а какие выходят за ее границы. Вы используете диаграммы классов, чтобы специфицировать эти абстракции и их предназначение.

Кооперации обсуждаются в главе 28.

2. Для моделирования простых кооперации.

Кооперация – это сообщество классов, интерфейсов и других элементов, которые работают в совокупности для формирования некоторого совместного поведения, которое не может быть обеспечено всеми этими элементами, взятыми в отдельности. Например, когда вы моделируете семантику транзакции в распределенной системе, то, глядя на отдельный класс, не сможете понять, что происходит. Эта семантика обеспечивается набором классов, работающих вместе. Диаграмма классов позволит визуализировать и специфицировать такой набор классов и их связей.

Свойство сохраняемости (устойчивости) обсуждается в главе 24, моделирование физических баз данных – в главе 30.

3. Для моделирования логической схемы базы данных.

Схему в данном контексте можно рассматривать как проект концептуального дизайна базы данных. Во многих областях требуется сохранять информацию в реляционной или объектно-ориентированной базе данных. Их схемы удобно моделировать при помощи диаграмм классов.

Типичные приемы моделирования

Моделирование простых кооперации

Ни один класс не существует сам по себе. Каждый из них работает в кооперации с другими, чтобы обозначить некую общую семантику, которая недостижима при автономном использовании всех тех же классов. Таким образом, наряду со словарем системы вам понадобится обратить внимание на визуализацию, специфицирование и документирование различных способов взаимодействия сущностей из этого словаря. Используйте диаграммы классов для представления таких кооперации.

Механизмы, как правило, тесно связаны с вариантами использования (см. главу 17). Сценарии – это потоки событий в составе вариантов использования, как обсуждается в главе 16.

Чтобы смоделировать кооперацию, необходимо:

- Идентифицировать механизм, который вы собираетесь моделировать. Механизм представляет некоторую функциональную или поведенческую часть моделируемой системы, полученную в результате взаимодействия совокупности классов, интерфейсов и прочих сущностей.
- Для каждого механизма идентифицировать классы, интерфейсы и прочие кооперации, которые участвуют в данной кооперации, а также связи между этими сущностями.
- Использовать сценарии для тестирования всех этих сущностей. На данном этапе выявляются части модели, которые были пропущены, а также те, которые семантически неверны.

- Убедиться, что все элементы наполнены правильным содержимым. Что касается классов, для начала обеспечьте оптимальный баланс обязанностей. Затем со временем проработайте конкретные атрибуты и операции.

На рис. 8.2 в качестве примера показан набор классов, описывающих реализацию автономного робота. Акцент сделан на классы, связанные с механизмом перемещения робота по заданному пути. Вы найдете здесь один абстрактный класс `Motor` (Мотор) с двумя конкретными потомками – `SteeringMotor` (МоторПоворотногоМеханизма) и `MainMotor` (ГлавныйМотор). Оба потомка наследуют от своего родителя `Motor` пять операций. И оба они, в свою очередь, представлены как часть другого класса – `Driver` (Привод). Класс `PathAgent` (АгентТраектории) имеет ассоциацию «один-к-одному» с `Driver` и «один-ко-многим» – с `CollisionSensor` (ДатчикСтолкновений). Никаких атрибутов и операций для `PathAgent` не показано, хотя его обязанности и заданы.

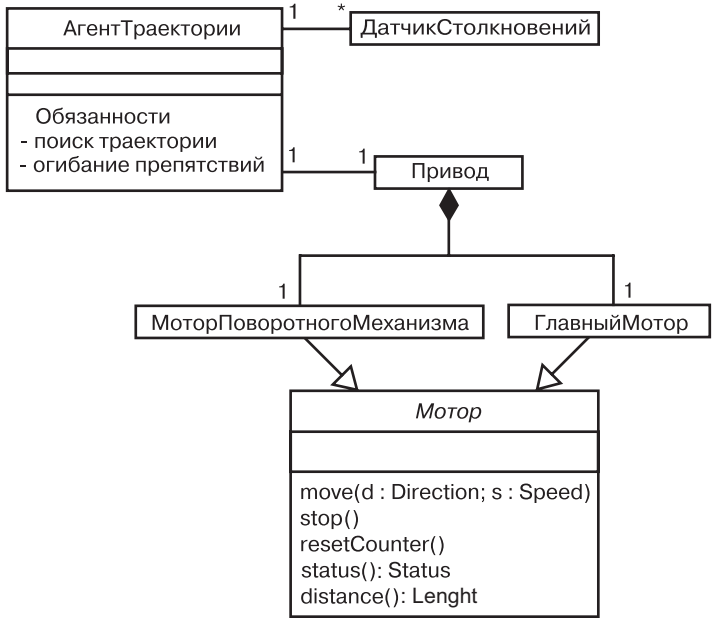


Рис. 8.2. Моделирование простой кооперации

В этой системе участвует много других классов, но диаграмма, которую мы рассматриваем, сосредоточена только на перемещении робота. В то же время некоторые классы, представленные на ней, можно найти и на других диаграммах. Например, хотя это здесь и не показано, класс `PathAgent` кооперируется по меньшей мере с двумя другими: `Environment` (Окружение) и `GoalAgent` (АгентЦели) в

механизме более высокого уровня, управляющем разрешением конфликтных ситуаций, в которых может оказаться робот. Классы `CollisionSensor` и `Driver`, а также их части кооперируются с классом `FaultAgent` (АгентОтказа) в составе механизма, отвечающего за непрерывную диагностику оборудования робота на предмет разнообразных ошибок. Фокусируя внимание на каждой из этих кооперациий в разных диаграммах, вы создаете понятное представление о системе с разных точек зрения.

Моделирование логической схемы базы данных

Моделирование распределения объектов обсуждается в главе 24, моделирование физической базы данных – в главе 30.

Многие системы, которые вам предстоит моделировать, будут иметь в своем составе сохраняемые объекты. Это значит, что они могут быть помещены в базу данных для последующего извлечения. Чаще всего вы будете использовать для хранения реляционные и объектно-ориентированные базы данных либо гибридные объектно-реляционные базы. UML хорошо подходит для моделирования как логических, так и физических схем баз данных.

Диаграммы классов UML представляют собой надмножество диаграмм «сущность-связь» (entity-relationship, E-R) – общего инструмента моделирования, используемого в логическом проектировании баз данных. В то время как классические E-R диаграммы сосредоточены только на данных, диаграммы классов идут на шаг вперед, позволяя также моделировать поведение. В физической базе данных эти логические операции обычно представлены в виде триггеров и хранимых процедур.

Чтобы смоделировать схему, необходимо:

Стереотипы обсуждаются главе 6.

- Идентифицировать те классы в модели, состояние которых не должно быть зависимым от времени жизни работающего приложения.
- Создать диаграмму классов, содержащую классы, выявленные на первом этапе. Вы можете определить ваш собственный набор стереотипов и помеченных значений, чтобы представить специфичные для базы данных детали.
- Раскрыть структурные подробности этих классов. В основном это означает необходимость подробно специфицировать их атрибуты, а также ассоциации с указанием множественности, которые связывают данные классы.
- Выявить общие образцы, которые усложняют физическое проектирование базы данных, например циклические ассоциации и ассоциации «один-к-одному». При необходимости создать промежуточные абстракции для упрощения логической структуры.

- Рассмотреть поведение классов, отмеченных на первом этапе, раскрыв операции, существенные для доступа к данным и обеспечения их целостности. Вообще, чтобы четче разграничить различные аспекты системы, следует инкапсулировать бизнес-правила, описывающие манипуляции наборами этих объектов, в слое, находящемся над данными устойчивыми классами.
- Там, где возможно, использовать инструментальные средства, которые помогут трансформировать логический дизайн в физический.

На заметку. Обсуждение логического проектирования базы данных выходит за рамки данной книги. Здесь мы просто показываем, как вы можете моделировать схемы при помощи UML. На практике вы чаще всего будете иметь дело со стереотипами, настроенными под используемую вами базу данных (реляционную или объектно-ориентированную).

На рис. 8.3 показан набор классов, описывающих информационную систему учебного заведения. Этот рисунок развивает диаграмму классов, которую мы рассматривали ранее (см. рис. 5.9 и 5.11), и показывает детали классов, существенных для конструирования физической схемы базы данных. Начиная с нижнего левого угла диаграммы вы найдете классы `Student` (Студент), `Course` (Курс) и `Instructor` (Преподаватель). Существует ассоциация между `Student` и `Course`, указывающая, что студенты посещают курсы. Более того, каждый студент может выбирать неограниченное число курсов; каждый курс может посещать множество студентов.

Моделирование примитивных типов обсуждается главе 4; агрегация – в главах 5 и 10.

Эта диаграмма раскрывает атрибуты всех шести классов. Отметим, что все атрибуты – примитивных типов. Когда вы моделируете схему базы, обычно связи с непримитивными типами моделируются в виде явных ассоциаций, а не атрибутов.

Два класса – `School` (Учебное заведение) и `Department` (Факультет) – включают несколько операций, предназначенных для манипулирования их частями. Эти операции важны для поддержания целостности данных: например, добавление и удаление факультетов может повлечь за собой некоторые недоразумения. Есть много других операций, которые стоит рассмотреть для этих и других классов (скажем, запрос определенной информации перед записью студента на курс). Такие функции можно трактовать скорее как бизнес-правила, а не операции, призванные обеспечивать целостность данных, поэтому их лучше поместить на более высокий уровень абстракции.

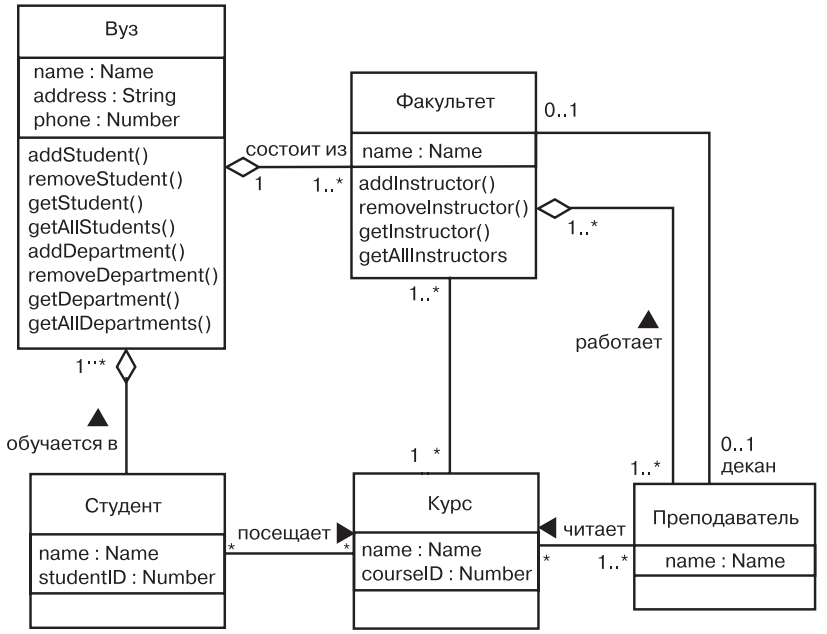


Рис. 8.3. Моделирование схемы базы данных

Прямое и обратное проектирование

Общие принципы и назначение моделирования обсуждаются в главе 1.

Как бы ни было важно моделирование, вы должны помнить, что главный продукт разработчиков – все-таки программное обеспечение, а не диаграммы. Конечно, назначение модели состоит в том, чтобы представить ПО, которое своевременно удовлетворяет изменяющиеся нужды его пользователей и требования бизнеса. По этой причине важно, чтобы создаваемые модели и их практическое воплощение четко соответствовали друг другу, причем с минимумом затрат (а лучше – совсем без таковых) на поддержание их в синхронизированном виде.

Диаграммы деятельности обсуждаются в главе 20.

В ряде случаев модели, которые вы создаете, вообще не отражаются в коде. Например, если вы моделируете бизнес-процесс, используя диаграмму деятельности, то многие ее элементы (деятельности) касаются людей, а не компьютеров. А иногда задача заключается в моделировании системы, части которой на выбранном вами уровне абстракции представляют собой элементы аппаратного обеспечения (хотя на другом уровне абстракции лучше показать совокупность компьютеров и программного обеспечения).

И все же разрабатываемые вами модели чаще преобразуются в код. UML не регламентирует никакого конкретного отображения на конкретный объектно-ориентированный язык программирования,

но проектировался, подразумевая такую возможность. Особенно это касается диаграмм классов, содержимое которых ясно отображается на все распространенные объектно-ориентированные языки программирования, в частности Java, C++, Smalltalk, Eiffel, Ada, ObjectPascal и Forte. Кроме того, UML предусматривает отображение на коммерческие объектные языки, такие как Visual Basic.

Стереотипы и помеченные значения обсуждаются в главе 6.

На заметку. Отображение UML на конкретные языки реализации для прямого и обратного проектирования выходит за круг тем, обсуждаемых в этой книге. На практике вам придется иметь дело со стереотипами и помеченными значениями, настроенными специально под используемый вами язык программирования.

Прямое проектирование (forward engineering) – это процесс трансформации модели в код посредством отображения на язык реализации. В результате прямого проектирования происходит потеря информации, поскольку модели, описанные на UML, семантически богаче, чем любой современный объектно-ориентированный язык программирования. Фактически это главная причина существования моделей наряду с кодом. Структурные средства, например кооперации, и поведенческие, например взаимодействия, могут быть ясно визуализированы с помощью UML, но не так ясно – в исходном коде.

Чтобы осуществить прямое проектирование диаграммы классов, необходимо:

- Идентифицировать конкретные правила отображения классов в код для выбранного вами языка (или языков) реализации. Эти правила будут касаться проекта в целом либо всей вашей организации.
- Если требуется, в зависимости от семантики выбранных языков установить ограничения на некоторые средства UML. Например, UML позволяет моделировать множественное наследование, а Smalltalk допускает только одиночное. Вы можете либо запретить разработчикам моделировать множественное наследование (что делает вашу модель зависимой от языка), либо разработать идиому, которая трансформирует эти богатые средства в язык реализации (что несколько усложняет отображение).
- Использовать помеченные значения для указания параметров реализации на вашем целевом языке программирования. Вы можете делать это на уровне индивидуальных классов, если нужен тонкий контроль, или же на более высоком уровне (например, на уровне кооперации или пакетов).
- Использовать инструменты для генерации кода.

Образцы обсуждаются в главе 6.

На рис. 8.4 вы видите простую диаграмму классов, представляющую реализацию *образца цепочки обязанностей* (responsibility pattern). Эта конкретная реализация включает три класса: `Client` (Клиент), `EventHandler` (ОбработчикСобытий) и `GUIEventHandler` (ОбработчикСобытийGUI)[4]. Классы `Client` и `EventHandler` – абстрактные, `GUIEventHandler` – конкретный. К `EventHandler` относится обычная операция, предусмотренная этим образцом, – `handleRequest` (обработатьЗапрос), хотя к его реализации добавлено два закрытых атрибута.

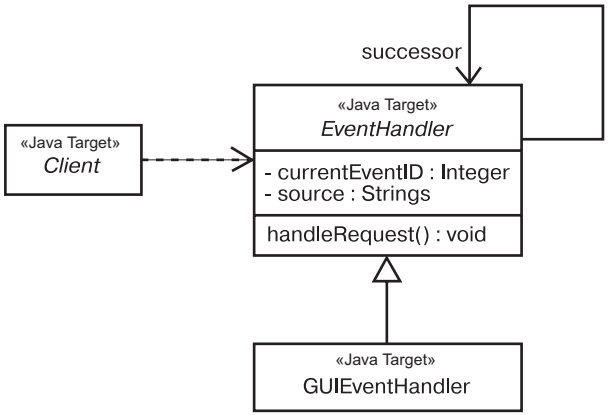


Рис. 8.4. Прямое проектирование

Все эти классы предполагают отображение на язык Java, как отмечено в их стереотипе. Прямое проектирование классов данной диаграммы на Java достаточно просто при использовании инструментальных средств. Прямое проектирование класса `EventHandler` на Java генерирует следующий код:

```
public abstract class EventHandler {

  EventHandler successor;
  private Integer currentEventID;
  private String source;

  EventHandler() {}
  public void handleRequest() {}
}
```

Обратное проектирование (reverse engineering) – это процесс трансформации кода в модель. Обратное проектирование порождает избыток информации, часть которой представлена на более низком уровне детализации, чем нужно для построения удобной

[4] Graphical User Interface – графический интерфейс пользователя. – *Прим. ред.*

модели. В то же время обратное проектирование неполно: при прямом проектировании модели в код происходит потеря информации, поэтому невозможно точно восстановить модель из кода, если только используемый вами инструмент не кодирует информацию в виде комментариев к исходному коду, которые выходят за пределы семантики языка реализации.

Чтобы осуществить обратное проектирование диаграммы классов, необходимо:

- Идентифицировать правила отображения для выбранного языка или языков реализации – это вам понадобится сделать для всего проекта или всей организации.
- Используя какое-либо инструментальное средство, указать код, который нужно подвергнуть обратному проектированию. Применяйте инструмент для генерирования новой модели или модификации существующей, в отношении которой ранее применялось прямое проектирование. Не стоит ожидать, что обратное проектирование породит единственную компактную модель на основе большого фрагмента кода. Вам придется выбирать небольшие фрагменты кода и строить из них модель.
- Просматривая модель, создать диаграмму классов с помощью какого-либо инструментального средства. Например, вы можете начать с одного или нескольких классов, а затем расширять диаграмму, прорабатывая конкретные связи или включая соседние классы. Показывайте или скрывайте детали диаграммы в соответствии с вашими намерениями.
- Вручную добавить к модели информацию о проектировании, чтобы показать те аспекты дизайна, которые пропущены или скрыты в исходном коде.

Советы и подсказки

Создавая диаграммы классов на UML, помните, что каждая из них – это лишь графическое изображение статического представления дизайна системы. Ни одна диаграмма классов не обязана включать все, что касается представления дизайна системы. Однако диаграммы классов в совокупности сообщают пользователю полную информацию, необходимую для статического представления системы; хотя каждая из них представляет только один ее аспект.

Хорошо структурированная диаграмма классов:

- сфокусирована на одном аспекте статического представления дизайна системы;
- содержит только те элементы, которые существенны для понимания данного аспекта;
- обеспечивает детализацию, соответствующую уровню абстракции диаграммы, включая только дополнения, важные для понимания;
- не настолько упрощена, чтобы создать у читателя ложное представление о важной семантике.

Когда вы рисуете диаграмму классов:

- присваивайте ей имя, соответствующее ее назначению;
- избегайте пересечения линий или хотя бы минимизируйте его;
- организуйте элементы так, чтобы семантические близкие сущности располагались рядом;
- используйте примечания и цвета в качестве меток, акцентирующих внимание на важных деталях диаграммы;
- постарайтесь не показывать слишком много видов связей. Вообще, на каждой диаграмме классов должен доминировать только один вид связей.

Часть III

Расширенное структурное моделирование

Глава 9. Расширенные классы

В этой главе:

- Классификаторы, особенности атрибутов и операций, разные виды классов
- Моделирование семантики класса
- Выбор правильного типа классификатора

Классы – это наиболее важные строительные блоки любой объектно-ориентированной системы. Однако они представляют собой всего лишь одну из разновидностей более общих строительных блоков UML – классификаторов. Классификатором называется механизм описания структурных и поведенческих свойств элемента системы. К классификаторам относятся классы, интерфейсы, типы данных, сигналы, компоненты, узлы, варианты использования и подсистемы.

Общие свойства классов обсуждаются в главе 4.

Классификаторы (и в особенности классы) помимо простых свойств – атрибутов и операций, описанных в части II данной книги, – имеют много расширенных. Вы можете моделировать множественность, видимость, сигнатуры, полиморфизм и другие характеристики. В UML моделировать семантику класса можно так, чтобы описать его значение на любом уровне формализации по вашему усмотрению.

Существует несколько видов классификаторов и классов; важно выбрать такой, который наилучшим образом моделирует вашу абстракцию реального мира.

Введение

Архитектура систем обсуждается в главе 2.

В начале проектирования дома вы принимаете решение о строительных материалах. Сперва речь идет просто о дереве, камне или стали – это уровень детализации, достаточный для продолжения работ. Выбор материала диктуется требованиями проекта: например, сталь и бетон подходят для строительства в местностях, подверженных ураганам. Определенный вами на начальном этапе

материал будет диктовать последующие проектные решения – скажем, выбор между деревом и сталью повлияет на массу строения, которую нужно поддерживать.

Продолжая работать над проектом, вы будете уточнять базовые проектные решения и добавлять детали, существенные для инженера-проектировщика, чтобы он проверил безопасность конструкций, и для строителя, – чтобы выполнил сборку. Например, вы можете указать, что нужно использовать дерево определенного сорта, более устойчивое перед вредными насекомыми.

Обязанности обсуждаются в главе 6.

Все то же самое относится и к построению программных систем. На ранней стадии проекта достаточно включить в систему, к примеру, класс `Customer` (Покупатель), который будет выполнять определенные обязанности. По мере уточнения архитектуры и перехода к конструированию вам придется принимать решения относительно структуры и поведения класса (соответственно, его атрибутов и операций), которые необходимы для выполнения классом своих обязанностей. И наконец, когда вы доберетесь до исполняемой системы, вам понадобится смоделировать такие детали, как видимость отдельных атрибутов и операций, семантика параллелизма класса в целом и его отдельных операций, а также интерфейсы, которые он реализует.

Прямое и обратное проектирование обсуждается в главах 8, 14, 18, 19, 20, 25, 30 и 31.

UML предусматривает представление множества расширенных свойств, как показывает рис. 9.1. Эта нотация позволяет вам визуализировать, специфицировать, конструировать и документировать классы на любом уровне детализации, достаточном для поддержки прямого и обратного проектирования моделей и кода.

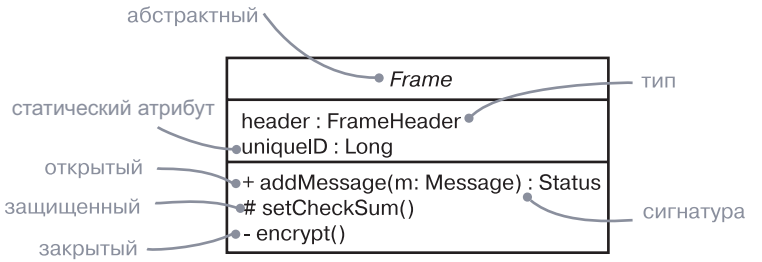


Рис. 9.1. Расширенные классы

Базовые понятия

Классификатор (classifier) – это механизм, описывающий структурные и поведенческие свойства. К классификаторам относятся классы, ассоциации, интерфейсы, типы данных, сигналы, компоненты, узлы, варианты использования и подсистемы.

Классификаторы

Моделирование словаря системы обсуждается в главе 4, дихотомия класс/объект – в главе 2.

В процессе моделирования вы исследуете абстракции, представляющие сущности реального мира и сущности, являющиеся составными частями вашего решения. Например, если вы строите Web-ориентированную систему обработки заказов, то словарь вашего проекта, вероятно, будет включать классы `Customer` (Покупатель) и `Transaction` (Транзакция). Первый описывает людей, заказывающих продукцию, а второй реализует артефакт, представляющий атомарное действие. В развернутой системе вы можете ввести компонент `Pricing` (Цены) с экземплярами, находящимися в каждом клиентском коде. У каждой из этих абстракций будут свои экземпляры. Разделение сущностей и экземпляров сущностей – важная часть моделирования.

Экземпляры обсуждаются в главе 13, пакеты – в главе 12, обобщения – в главах 5 и 10, сообщения – в главе 16, интерфейсы – в главе 11, типы данных – в главах 4 и 11, сигналы – в главе 21, компоненты – в главе 15, узлы – в главе 27, варианты использования – в главе 17, подсистемы – в главе 32.

Некоторые элементы в UML – например, пакеты или связи обобщения – не имеют экземпляров. Те же элементы моделирования, которые имеют экземпляры, являются классификаторами. Но более важный их признак – наличие как структурных свойств (в форме атрибутов), так и поведенческих (в форме операций). Все экземпляры классификатора обладают общими свойствами, но при этом каждый имеет свое собственное значение для каждого атрибута.

Наиболее важный вид классификаторов в UML – это *класс*, представляющий собой описание набора объектов с одинаковыми атрибутами, операциями, связями и семантикой. Однако классы – не единственная разновидность классификаторов. UML представляет множество других видов, немаловажных для моделирования:

- **интерфейс** – набор операций, используемых для спецификации сервиса класса или компонента;
- **тип данных** – тип, значения которого неизменны. Примеры: примитивные встроенные типы (числа, строки и др.), типы перечислений (Boolean и др.).
- **ассоциация** – описание набора ссылок, каждая из которых соединяет ряд объектов;
- **сигнал** – спецификация асинхронного сообщения, передаваемого между экземплярами;
- **компонент** – модульная часть системы, скрывающая свою реализацию за набором внешних интерфейсов;
- **узел** – физический элемент, существующий во время исполнения и представляющий вычислительный ресурс, который обычно наделен по меньшей мере некоторой памятью и часто – вычислительными возможностями;
- **вариант использования** – описание последовательности действий (включая их разновидности), осуществляемых системой и порождающих значимый результат для определенного действующего лица;
- **подсистема** – компонент, представляющий главную часть системы.

Как правило, каждый вид классификаторов может иметь как структурные, так и поведенческие свойства. Более того, при моделировании системы с помощью любого из этих классификаторов вы можете использовать все расширенные свойства, описанные в настоящей главе, чтобы обеспечить уровень детализации, достаточный для передачи смысла абстракции.

Графически UML представляет все эти виды классификаторов так, как показано на рис. 9.2.

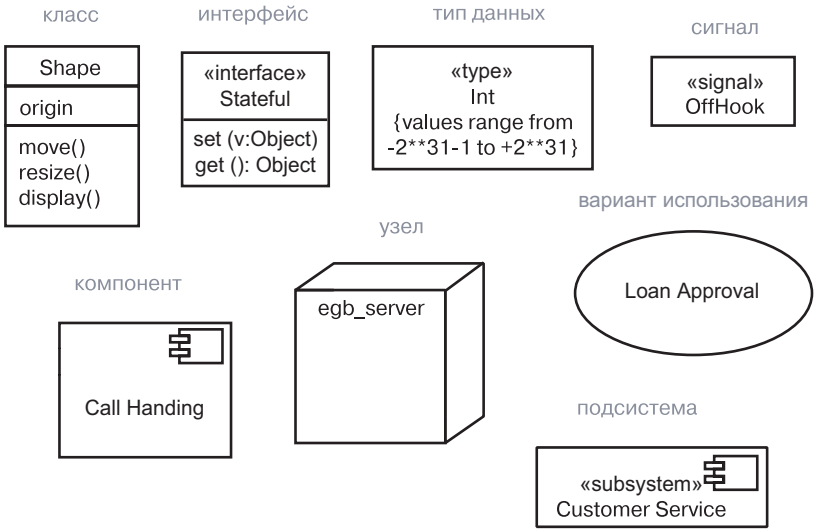


Рис. 9.2. Классификаторы

На заметку. Минималистский подход потребовал бы использования одной пиктограммы для всех видов классификаторов, а тенденция к максимализму, – наоборот, использования особой пиктограммы для каждого вида. Оба эти варианта не имеют смысла, потому что, с одной стороны, к классификаторам относится слишком много элементов, а с другой, некоторые из них, например классы и типы данных, не так уж сильно различаются. В UML соблюдается разумный баланс: ряд классификаторов имеет свои собственные пиктограммы, а для различения других используются специальные ключевые слова (в частности, `type`, `signal` и `subsystem`).

Видимость

Видимость (visibility) – одна из деталей дизайна, которую вы можете определить для атрибута или операции. Это свойство, указывающее на возможность использования данного атрибута или операции другими классификаторами. В UML можно специфицировать четыре уровня видимости:

Классификатор может видеть другой классификатор, если он находится в области действия и если существует явная или неявная связь между ними; связи обсуждаются в главах 5 и 10; наследники происходят из связей обобщения, как описано в главе 5; права доступа позволяют классификатору разделять его закрытые свойства с другими, как описано в главе 10.

1. `public` (открытый). Любой внешний классификатор, которому виден данный, может использовать это свойство. Обозначается символом + перед именем атрибута или операции.
2. `protected` (защищенный). Любой наследник классификатора может использовать данное свойство. Обозначается символом # (решетка) перед именем атрибута или операции.
3. `private` (закрытый). Данное свойство может использовать только сам классификатор. Обозначается символом – (дефис) перед именем атрибута или операции.
4. `package` (пакетный). Только классификаторы, объявленные в том же пакете, могут использовать данное свойство. Обозначается символом ~ (тильда) перед именем атрибута или операции.

На рис. 9.3 показана совокупность открытых, защищенных и закрытых атрибутов и операций класса `Toolbar`.

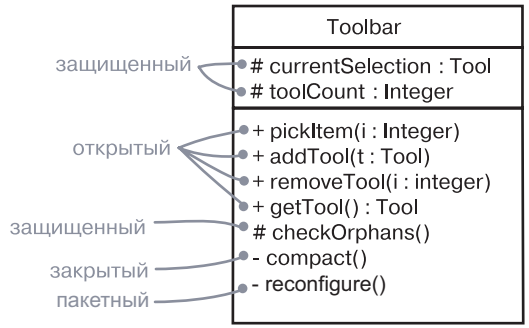


Рис. 9.3. Видимость

Указывая видимость средств классификатора, вы обычно скрываете детали его реализации и делаете видимыми только те свойства, которые необходимы для исполнения им своих обязанностей абстракции. Это основа скрытия информации, обеспечивающая построение прочной устойчивой системы. Если вы не снабжаете свойство символом видимости явным образом, то по умолчанию принимается значение `public`.

На заметку. Свойство видимости UML соответствует общепринятой семантике большинства языков программирования, включая C++, Java. Ada и Eiffel. Отметим, однако, что эти языки несколько различаются в представлении семантики видимости.

Область действия экземпляра и статическая область действия

Экземпляры обсуждаются в главе 13.

Еще одна важная деталь, которую вы можете задать для атрибутов и операций вашего классификатора, – *область действия* (scope). Область действия свойства указывает на то, может ли каждый экземпляр данного классификатора иметь собственное значение этого свойства либо должно существовать только одно его значение, разделенное всеми экземплярами классификатора. В UML существуют два типа области действия:

1. `instance` – область действия экземпляра. Каждый экземпляр классификатора имеет собственное значение свойства. Этот вариант принят по умолчанию и не требует дополнительной нотации;
2. `static` – статическая область действия, или область действия класса (class scope). Предусмотрено только одно значение свойства для всех экземпляров классификатора.

Рис. 9.4 – это упрощенный вариант рис. 9.1. Здесь показан атрибут со статической областью действия, о чем свидетельствует подчеркивание его имени. Для отображения области действия экземпляра не используются никакие дополнения.

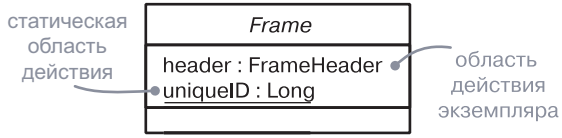


Рис. 9.4. Область действия

Вообще говоря, большинство свойств (атрибутов и операций) классификаторов, которые вам предстоит моделировать, будут иметь область действия экземпляра. В статической области действия находятся закрытые атрибуты, которые должны быть разделены между экземплярами класса (например, для генерации уникальных идентификаторов новых экземпляров класса).

На заметку. Статическая область действия соответствует статическим атрибутам и операциям в языках C++ и Java.

Статическая область действия применительно к операциям ведет себя несколько иначе. Операция экземпляра имеет неявный параметр, который указывает на объект, подлежащий обработке. У статической операции нет такого параметра – она подобна традиционной глобальной процедуре без целевого объекта. Статические операции используются для создания экземпляров или манипулирования статическими атрибутами.

Абстрактные, листовые и полиморфные элементы

Обобщение обсуждается в главах 5 и 10, экземпляры – в главе 13.

Связи обобщения используются для моделирования структуры классов с более общими абстракциями, расположенными на вершине иерархии, и более детальными – внизу. В пределах такой иерархии некоторые классы часто определяются как абстрактные – это значит, что они не могут иметь прямых экземпляров. Например, на рис. 9.5 `Icon` (Пиктограмма), `RectangularIcon` (ПрямоугольнаяПиктограмма) и `ArbitraryIcon` (ПиктограммаПроизвольнойФормы) – абстрактные классы. В отличие от них, конкретные классы `Button` (Кнопка) и `OKButton` (КнопкаОК) могут сопровождаться экземплярами.

Создавая новый класс, вы, вероятно, захотите, чтобы он наследовал свои свойства от других, более общих классов, а также имел потомков – более специализированные классы, наследующие его свойства. Это нормальная семантика классов в UML. Впрочем, может случиться и так, что класс не должен иметь потомков. Такой элемент называется *листовым* и обозначается в UML свойством `leaf`, записанным под именем класса. Например, на рис. 9.5 `OKButton` – листовой класс.

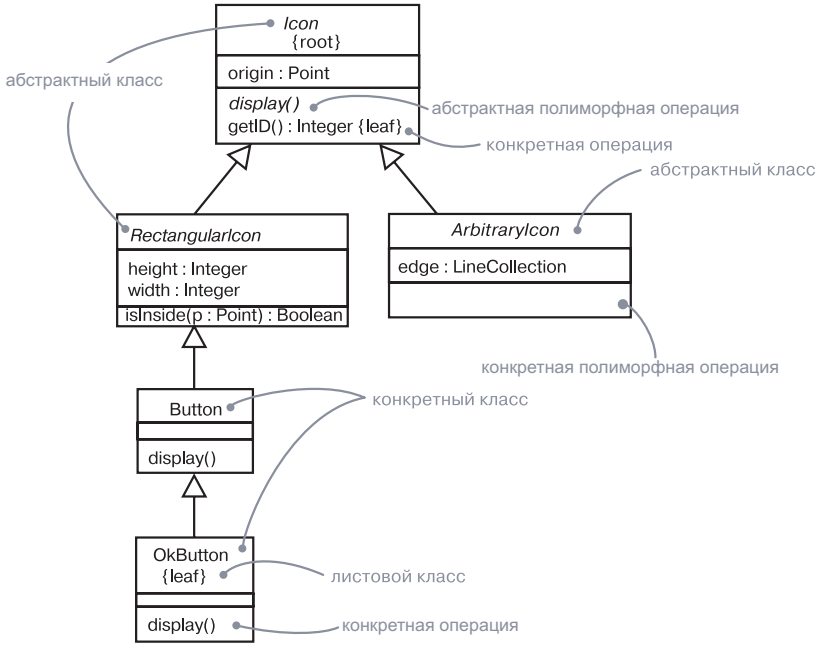


Рис. 9.5. Абстрактные и конкретные классы и операции

Сообщения обсуждаются в главе 16.

Операции имеют аналогичные свойства. Обычно операция *полиморфна*, то есть вы можете специфицировать операцию с той же сигнатурой в разных местах иерархии классов. Операция в дочернем классе отменяет поведение такой же операции родительского класса (приоритетна перед ней). Когда сообщение посылается во время исполнения, конкретная вызываемая операция выбирается из иерархии полиморфно – определяется во время исполнения соответственно типу объекта. Например, операции `display` (отобразить) и `isInside` (внутри) на рис. 9.5 полиморфны. Более того, операция `Icon::display()` абстрактна: она не реализована в классе `Icon` и требует от потомков предоставления собственных ее реализаций. В UML имена абстрактных операций по аналогии с абстрактными классами выделяются курсивом. А вот `Icon::getID()` – листовая операция, на что указывает слово `leaf`. Следовательно, она не полиморфна и не может быть переопределена в классах-потомках. В языке Java такие операции называются `final` (конечными).

На заметку. Абстрактные операции UML в языке C++ называются пустыми виртуальными (pure virtual), листовые операции UML – невиртуальными (nonvirtual), а в Java – конечными (final).

Множественность

Экземпляры обсуждаются в главе 13.

Множественность также является свойством ассоциаций (см. главы 5 и 10). Атрибуты связаны с семантикой ассоциаций, как показано в главе 10.

Всякий раз при использовании класса имеет смысл предположить, что число его экземпляров не ограничено (если только он не является абстрактным и при нем не может быть непосредственных экземпляров, хотя экземпляры его потомков могут существовать в любом количестве). Иногда, однако, возникает необходимость ограничить количество экземпляров класса, а еще чаще требуется указать, что оно может равняться нулю (когда перед вами класс-утилита, который представляет только статические операции и атрибуты), единице (класс-одиночка), определенному числу либо неопределенному множеству (по умолчанию).

Возможное число экземпляров класса называется множественностью. Иными словами, *множественность* (multiplicity) – это диапазон допустимых значений количества сущностей. В UML вы можете указать множественность класса в выражении, находящемся в правом верхнем углу пиктограммы класса. Например, на рис. 9.6 `NetworkController` (КонтроллерСети) – *класс-одиночка* (singleton). Кроме того, в системе должно быть ровно три экземпляра класса `ControlRod` (РегулирующийСтержень).

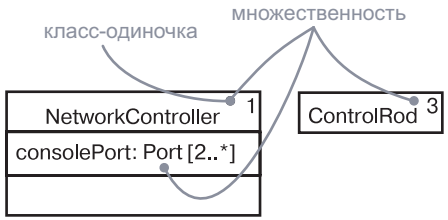


Рис. 9.6. Множественность

Множественность устанавливается и для атрибутов: достаточно написать соответствующее выражение в квадратных скобках сразу после имени атрибута. Так, из рис. 9.6 явствует, что внутри экземпляра `NetworkController` может быть несколько экземпляров `consolePort`.

На заметку. Множественность класса имеет смысл в определенном контексте. На самом верхнем уровне это контекст всей системы. Вся система может рассматриваться как структурированный классификатор.

Атрибуты

На самом абстрактном уровне при моделировании структурных свойств класса, то есть его атрибутов, вы просто пишете имена этих атрибутов. Обычно такой информации достаточно, чтобы среднестатистический пользователь понял назначение вашей модели. Однако можно ко всему прочему специфицировать видимость, область действия и множественность (см. предыдущие разделы), а также тип, начальное значение и изменяемость каждого атрибута.

С целью различения наборов взаимосвязанных атрибутов (таких как служебные атрибуты) можно применять стереотипы – см. главу 6.

Полный синтаксис атрибута в UML выглядит так:

```
[видимость] имя
[':' тип] ['[' множественность ']']
['=' начальное значение]
[строка свойств {, строка свойств}]
```

Приведем несколько примеров корректных объявлений атрибута:

- `origin` — только имя
- `+ origin` — видимость и имя
- `origin : Point` — имя и тип
- `name : String[0..1]` — имя, тип и множественность
- `origin : Point = (0, 0)` — имя, тип и начальное значение
- `id : Integer {readonly}` — имя и свойство

Атрибуты всегда изменяемы, если только явно не указано другое. Если требуется сообщить, что значение атрибута не может быть изменено после инициализации объекта, используйте свойство `readonly`. В основном это понадобится вам при моделировании констант или атрибутов, которые инициализируются при создании экземпляра и после этого не изменяются.

На заметку. Свойство readonly соответствует свойству const в C++.

Операции

Сигнатуры обсуждаются в главе 21.

На самом высоком уровне абстракции при моделировании поведенческих свойств класса, то есть операций и сигналов, вы просто указываете имя каждой операции. Обычно этой информации достаточно, чтобы среднестатистический пользователь понял назначение модели. Однако можно также специфицировать видимость и область действия каждой операции (как обсуждалось в предыдущих разделах), а кроме того, ее параметры, тип возвращаемого значения, семантику параллелизма и прочие свойства. Имя операции вместе с ее параметрами, включая тип возвращаемого значения, если оно есть, называется *сигнатурой* операции.

На заметку. Понятия «операция» и «метод» в UML различаются. Операция – это сервис, который может быть запрошен у любого объекта класса для реализации поведения, а метод – реализация операции. Каждой неабстрактной операции класса должен быть сопоставлен метод, который содержит исполняемый алгоритм в виде тела класса (обычно реализованного на некотором языке программирования или в виде структурированного текста). В структуре наследования допускается множество методов для одной и той же операции, и полиморфизм позволяет указать, какой метод из иерархии вызывается во время исполнения.

Для обозначения наборов связанных операций (таких как вспомогательные функции) можно использовать и стереотипы – см. главу 6.

Полный синтаксис операции в UML выглядит так:

```
[видимость] имя ['(' список параметров ')']
[':' тип возвращаемого значения]
[строка свойств {, строка свойств}]
```

Приведем в качестве примера несколько корректных объявлений операции:

- `display` — только имя
- `+ display` — видимость и имя

- `set(n : Name, s : String)` имя и параметры
- `getID() : Integer` имя и тип возврата
- `restart() {guarded)` имя и свойство

В сигнатуре операции вы можете указать ноль, один или несколько параметров, каждый из которых выражается следующим синтаксисом:

```
[direction] name : type [=default-value]
```

Направление (`direction`) может принимать одно из следующих значений:

- `in` входной параметр, не может быть модифицирован
- `out` выходной параметр, может быть модифицирован для передачи информации вызывающему коду
- `inout` входной параметр, может быть модифицирован для передачи информации вызывающему коду

На заметку. Параметры `out` и `inout` эквивалентны соответственно возвращаемому параметру и параметру `in`. Значения `out` и `inout` предусмотрены для совместимости с более старыми языками программирования. Используйте вместо них явные возвращаемые параметры.

В дополнение к свойствам `leaf` и `abstract`, описанным выше, есть другой ряд свойств, характеризующих операции:

1. `query` (запрос): исполнение операции оставляет состояние системы без изменений. Другими словами, операция представляет собой простую функцию, которая не имеет побочных эффектов;
2. `sequential` (последовательная): вызывающий код должен быть скоординирован вне объекта, чтобы существовал только один поток в объекте в единицу времени. При наличии множества потоков управления семантика и целостность объекта не гарантируются;
3. `guarded` (защищенная): семантика и целостность объекта гарантируются при наличии множества потоков управления за счет выстраивания последовательности всех вызовов защищенных операций объекта. В результате только одна операция объекта может быть вызвана в единицу времени, что обуславливает последовательную семантику;
4. `concurrent` (параллельная): семантика и целостность объекта гарантируются при наличии множества потоков управления за счет того, что операция трактуется как атомарная. Множественные вызовы из параллельных потоков управления могут осуществляться одновременно для одного объекта на любых параллельных операциях, причем все должны обрабатываться параллельно, с корректной семантикой. Параллельные операции следует спроектировать так, чтобы они корректно выполнялись при наличии параллельной, последовательной и защищенной операции на одном и том же объекте;
5. `static` (статическая): операция не имеет неявного параметра – целевого объекта и ведет себя подобно традиционной глобальной процедуре.

Активные объекты, процессы и потоки обсуждаются в главе 23. Базовые свойства классов обсуждаются в главе 4.

Свойства параллелизма (`sequential`, `guarded` и `concurrent`) предназначены для поддержки семантики параллелизма операций. Эти свойства существенны только в присутствии активных объектов, процессов или потоков.

Шаблонные классы

Шаблон – это параметризованный элемент. В таких языках, как C++ и Ada, вы можете создавать шаблонные классы, каждый из которых определяет семейство классов (также возможно создание шаблонов функций, каждый из которых определяет семейство функций). Шаблоны могут включать слоты для классов, объектов и значений; эти слоты служат параметрами шаблонов. Вы не можете использовать шаблон сам по себе: сначала следует создать его экземпляр. Этот процесс предполагает связывание формальных параметров шаблона с реальными. В результате для шаблонного класса создается конкретный, который можно использовать как любой другой обыкновенный класс.

Чаще всего шаблоны классов применяются для описания контейнеров, экземпляры которых могут быть созданы для хранения специфических элементов, что обеспечивает для них безопасность типов. Например, следующий фрагмент кода C++ объявляет параметризованный класс `Map` (Карта):

```
template<class Item, class VType, int Buckets>
class Map {
public:
 virtual map(const Item&, const &VType);
 virtual Boolean isMappen(const Item&) const;
...
};
```

Затем вы можете создать экземпляр этого шаблона для отображения объектов `Customer` (Покупатель) на объекты `Order` (Заказ):

```
m : Map<Customer, Order, 3>;
```

UML тоже позволяет моделировать шаблонные классы. Как показано на рис. 9.7, они изображаются так же, как обычные, но с дополнительной пунктирной рамочкой в верхнем правом углу пиктограммы класса, где перечисляются параметры шаблона.

Зависимости обсуждаются в главах 5 и 10, стереотипы – в главе 6.

Как следует из рисунка, моделировать создание экземпляра шаблонного класса можно двумя способами – явно и неявно. В первом случае вы объявляете класс, имя которого описывает связывание; во втором – применяете стереотип зависимости (такой как `bind`), который указывает на то, что источник создает экземпляр целевого шаблона, используя действительные параметры.

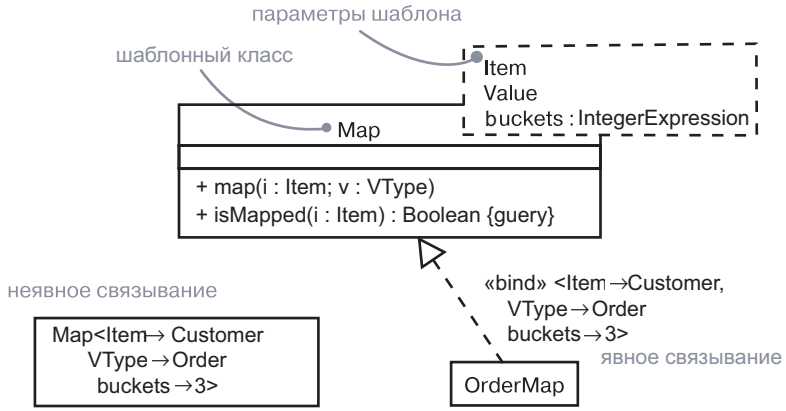


Рис. 9.7. Шаблонные классы

Стандартные элементы

Механизмы расширения UML обсуждаются в главе 6.

Все механизмы расширения UML применимы к классам. Чаще всего вы будете использовать помеченные значения для расширения свойств класса (такие как спецификация версии класса) и стереотипы для спецификации новых видов компонентов (в частности, компонентов, специфичных для модели).

UML определяет четыре стандартных стереотипа, прилагаемых к классам:

1. `metaclass` – описывает классификатор, объектами которого являются все классы;
2. `powertype` – описывает классификатор, чьими объектами являются дочерние классы заданного родительского класса;
3. `stereotype` – указывает на то, что классификатор представляет собой стереотип, который может быть приложим к другим элементам;
4. `utility` – описывает класс, атрибуты и операции которого имеют статическую область действия.

На заметку. Многие стандартные стереотипы или ключевые слова, имеющие отношение к классам, обсуждаются в других главах (см. предметный указатель).

Типичные приемы моделирования

Моделирование семантики класса

Общее применение классов рассматривается в главе 4. Основы моделирования обсуждаются в главе 1. Семантику операции можно моделировать, помимо прочего, с помощью диаграмм активности (см. главу 20).

Главная цель использования классов – моделирование абстракций, которые определяются проблемой, которую вы пытаетесь решить, либо технологией, применяемой для решения данной проблемы. После того как эти абстракции идентифицированы, следующий шаг – спецификация их семантики.

UML предоставляет в ваше распоряжение широкий спектр средств моделирования, начиная с относительно неформальных (обязанностей) и заканчивая в высшей степени формализованными (OCL – Object Constraint Language). Имея такой выбор, вы должны определить желаемый уровень детализации в соответствии с коммуникативным предназначением вашей модели. Если ее назначение состоит в передаче информации конечным пользователям и экспертам в предметной области, она может быть менее формальной. Если же цель заключается в поддержке прямого и обратного проектирования между моделью и кодом, то модель должна быть формализована в большей степени. Если назначение модели – ее математическое представление со строгим доказательством корректности, следует максимально формализовать ее.

На заметку. Менее формальная модель – не значит менее точная. Просто она менее подробная. При выборе уровня формальности вы должны соблюдать меру, обеспечивая детализацию, достаточную для того, чтобы сделать модель вполне понятной.

Чтобы смоделировать семантику класса, выберите один из следующих вариантов, выстроенных в порядке возрастания формальности:

Обязанности обсуждаются в главе 4.

- определите обязанности класса. Обязанность – это контракт или обязательства типа или класса; она описывается в примечаниях, присоединенных к классу, либо в дополнительном разделе, включенном в пиктограмму класса;
- специфицируйте семантику класса как единого целого, используя структурированный текст, который представлен

в виде примечания со стереотипом semantics, присоединенного к классу;
- опишите тело каждого метода, используя структурированный текст или язык программирования, который представлен в форме примечания, соединенного с операцией связью зависимости;
- задайте пред- и постусловия каждой операции, а также инварианты всего класса как целого, используя структурированный текст. Эти элементы отображаются в примечаниях соответственно со стереотипами precondition, postcondition и invariant, соединенных с операциями или классами связью зависимости;
- специфицируйте для класса машину состояний (state machine). *Машина состояний*, или *конечный автомат*, – это поведение, описывающее последовательность состояний, которые принимает объект за время своего жизненного цикла в ответ на события, вместе с его реакцией на эти события;
- проработайте внутреннюю структуру класса;
- специфицируйте кооперацию, представляющую класс. Под *кооперацией* понимается сообщество ролей и других элементов, которые работают совместно для представления некоего общего поведения, отличного от поведения простой суммы этих элементов. Кооперация имеет как структурную, так и динамическую части, поэтому вы можете использовать ее для описания всех сторон семантики класса;
- сформулируйте пред- и постусловия каждой операции, а также инварианты всего класса как целого, средствами формального языка наподобие OCL.

Спецификация тела метода обсуждается в главе 3. Спецификация семантики операции обсуждается в главе 20, машины состояний обсуждаются в главе 22, кооперации – в главе 28, внутренние структуры – в главе 15. Язык OCL описан в справочнике «UML» (выходные данные см. во введении, раздел «Цели»).

На практике вы будете использовать для разных абстракций вашей системы некую комбинацию вышерассмотренных подходов.

На заметку. Моделируя семантику класса, помните о том, что вы намерены выразить в первую очередь: ЧТО делает класс, или же КАК он это делает. В первом случае спецификация семантики класса предполагает описание его открытого, внешнего представления, во втором –закрытого, внутреннего. Можно использовать сочетание обоих представлений, внешнее показывая клиентам класса, а внутреннее – тем, кто занят его реализацией.

Советы и подсказки

Когда вы моделируете классификаторы на UML, не забывайте о существовании широкого спектра строительных блоков, предоставленных в ваше распоряжение, – от интерфейсов до классов,

компонентов и др. Вы должны выбрать тот элемент, который наилучшим образом соответствует вашей абстракции. Хорошо структурированный классификатор наделен следующими характеристиками:

- имеет и структурные, и поведенческие аспекты;
- обладает высокой согласованностью и слабой связанностью;
- раскрывает только те свойства, которые необходимы клиентам, чтобы использовать класс, и скрывает все остальные;
- недвусмыслен в плане предназначения и семантики;
- специфицирован не настолько подробно, чтобы не оставить никакой свободы тем, кто его будет реализовывать;
- специфицирован не настолько поверхностно, чтобы допустить неоднозначное понимание.

Когда вы изображаете классификатор на UML:

- показывайте только те его свойства, которые важны для понимания абстракции в данном контексте;
- выбирайте версию со стереотипом, которая наилучшим образом отражает назначение данного классификатора визуально.

Глава 10. Расширенные связи

В этой главе:

- Расширенные связи: зависимости, обобщения, ассоциации, реализации и уточнения
- Моделирование систем связей
- Создание систем связей

Моделируя сущности, формирующие словарь вашей системы, вы должны также показать, как эти сущности связаны друг с другом. Связи, однако, могут быть достаточно сложными. Визуализация, специфицирование, конструирование и документирование систем связей требуют ряда расширенных свойств.

Зависимости, обобщения и ассоциации – наиболее важные строительные блоки связей в UML. Эти связи обладают множеством свойств помимо тех, что уже были описаны в предыдущей части. Также вы можете моделировать множественное наследование, навигацию, композицию, уточнение и другие характеристики. Используя четвертый вид связей – реализацию, можно моделировать соединение между интерфейсом и классом или компонентом либо между вариантом использования и кооперацией. UML допускает моделирование семантики связей на любом уровне формальности.

Управление сложными системами связей требует использования правильных связей на правильном уровне детализации, чтобы не имело место ни чрезмерное упрощение, ни излишнее усложнение вашей системы.

Введение

Варианты использования и сценарии обсуждаются в главе 17.

Когда вы проектируете дом, важно определить, как будут располагаться комнаты по отношению друг к другу. Скажем, вы решаете поместить главную ванную комнату на первом этаже, подальше от фасада дома. Дальше надлежит продумать общий план расположения комнат (например, учесть необходимость вноса продуктов из гаража на кухню: было бы неразумно, если бы на этом пути

пришлось проходить через ванную, поэтому такой вариант вы отвергнете).

Можно составить достаточно подробный план дома, просто учитывая эти основные связи и варианты использования. Однако если не учитывать в проекте более сложных связей, в конце концов вы столкнетесь с серьезными трудностями. Допустим, вас вполне устраивает расположение комнат на каждом этаже, но комнаты на разных этажах соседствуют неудачно. В частности, комнату вашей дочери вы поместили прямо над своей ванной, а девочка учится играть на барабане. Понятно, что изначальную схему придется пересмотреть.

Аналогичным образом вы должны установить, как скрытые механизмы дома (коммуникации, проводка, несущие конструкции и т.п.) могут повлиять на поэтажный план. Так, например, стоимость конструкции значительно возрастет, если вы не разместите комнаты таким образом, чтобы они имели общие стены, в которых можно проложить трубы и водостоки.

Прямое и обратное проектирование обсуждаются в главах 8, 14, 18, 19, 20, 25, 30 и 31.

То же самое происходит, когда вы создаете программное обеспечение. Зависимости, обобщения и ассоциации – наиболее общие связи, с которыми вы сталкиваетесь при моделировании программных систем, но вам понадобится немало расширенных средств для описания таких связей. В результате вам удастся охватить элементы многих систем, что позволит избежать недоработок в дизайне.

С этой целью UML обеспечивает представление множества дополнительных свойств (рис. 10.1). Данная нотация позволяет вам визуализировать, специфицировать и документировать системы связей на любом уровне детализации по вашему желанию, – даже достаточном для осуществления прямого и обратного проектирования моделей и кода.

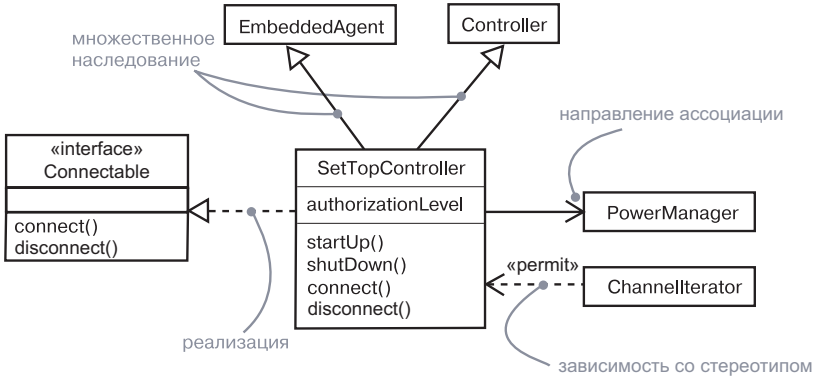


Рис. 10.1. Расширенные связи

Базовые понятия

Связь – это соединение сущностей. В объектно-ориентированном моделировании существуют четыре наиболее важных типа связей: зависимости, обобщения, ассоциации и реализации. Связи изображаются линиями разного начертания.

Зависимости

Базовые свойства зависимостей обсуждаются в главе 5.

Зависимость – это связь использования, указывающая, что изменение спецификаций одной сущности, например класса `SetTopController` (УстановитьВерхнийКонтроллер) на рис. 10.1, может повлиять на другие сущности, которые используют ее, например на класс `ChannelIterator` (ИтераторКанала, – но не наоборот. Зависимость изображается в виде пунктирной линии со стрелкой, направленной к той сущности, от которой зависит еще одна. Применяйте зависимости, когда хотите показать, что некая сущность использует другую.

Механизмы расширения UML обсуждаются в главе 6. Диаграммы классов обсуждаются в главе 8.

Простая, без дополнений, связь зависимости встречается чаще всего. Однако, если вы хотите подчеркнуть некие смысловые оттенки, UML предоставляет вам набор стереотипов, которые могут быть приложены к связи зависимости. Имеющиеся стереотипы можно разделить на несколько групп.

Во-первых, есть стереотип, который касается связей зависимости между классами и объектами на диаграмме классов:

1. `bind` – показывает, что исходный объект создает экземпляр целевого, используя заданные реальные параметры.

Шаблоны и зависимость `bind` *обсуждаются в главе 9.*

Используйте `bind`, когда нужно уточнить подробности, касающиеся шаблонных классов. Например, связь между шаблонным контейнерным классом и экземпляром этого класса должна моделироваться зависимостью `bind`. Она включает список действительных аргументов, соответствующих формальным аргументам шаблона.

2. `derive` – показывает, что источник может быть вычислен по цели.

Атрибуты обсуждаются в главах 4 и 9, ассоциации – в главе 5 и ниже в настоящей главе.

Использовать `derive` следует при моделировании связи между двумя атрибутами или двумя ассоциациями, одна из которых конкретна, а вторая – концептуальна. Например, класс `Person` (Человек) может иметь конкретный атрибут `BirthDate` (ДатаРождения), а также атрибут `Age` (Возраст), который происходит от `BirthDate`, поэтому не объявлен отдельно в классе. Связь между `BirthDate` и `Age` выражается зависимостью `derive`, то есть `Age` выводится из `BirthDate`.

3. `permit` – показывает, что источник имеет заданную видимость для цели.

Зависимости типа `permit` *обсуждаются в главе 5.*

Такая зависимость пригодится вам, если вы захотите обеспечить классу доступ к закрытым свойствам другого класса. В C++ для этих же целей предусмотрены классы-друзья (friends).

4. `instanceOf` – показывает, что исходный объект является экземпляром целевого. Обычно отображается в текстовой нотации вида `исходный объект : Целевой объект`.
5. `instantiate` – показывает, что источник создает экземпляры цели.

Дихотомия «класс/объект» обсуждаются в главе 2.

Два последних стереотипа позволяют вам явно моделировать связи «класс/объект». Можно использовать `instanceOf` при моделировании связи между классом и объектом на одной и той же диаграмме либо связи между классом и его метаклассом; однако последняя обычно отображается в текстовом синтаксисе. Стереотип `instantiate` свидетельствует о том, что один класс создает объекты другого.

6. `powertype` – сообщает, что целевой объект является супертипом исходного. Супертип – это классификатор, объекты которого являются дочерними по отношению к заданному родительскому.

Логическое моделирование баз данных обсуждается в главе 8, моделирование физических баз данных – в главе 30.

Следует использовать `powertype`, когда нужно моделировать те классы, которые по отношению к другим являются классификаторами (как иногда бывает при моделировании баз данных).

7. `refine` – показывает, что источник находится на более тонком уровне абстракции, чем цель.

Используется при моделировании классов, представляющих одно и то же понятие на разных уровнях абстракции. Например, в процессе анализа вы можете определить класс `Customer` (Покупатель), который в процессе проектирования будет уточнен до более детализированного класса `Customer` и завершен реализацией.

8. `use` – специфицирует, что семантика исходного элемента зависит от семантики открытой части целевого.

Применяйте `use`, когда хотите явно обозначить зависимость как связь использования, в противовес прочим значениям зависимостей, представленным в стереотипах.

Пакеты обсуждаются в главе 12.

Упомянем два стереотипа, касающихся связей зависимости между пакетами:

1. `import` – показывает, что открытое содержимое целевого пакета входит в открытое пространство имен исходного пакета, как если бы они были декларированы в исходном;
2. `access` – показывает, что открытое содержимое целевого пакета входит в закрытое пространство имен исходного. Неквалифицированные имена могут применяться внутри исходного, но не могут реэкспортироваться.

Используйте import и access, когда захотите задействовать элементы, объявленные в других пакетах. Импорт элементов позволяет отказаться от применения полных квалифицированных имен для ссылок на элементы других пакетов в текстовых выражениях.

Варианты использования обсуждаются в главе 17.

Два стереотипа относятся к связям использования между вариантами использования:

1. extend – показывает, что целевой вариант использования расширяет поведение исходного;
2. include – показывает, что исходный вариант использования явно включает в себя поведение другого варианта использования в месте, указанном в исходном.

Используйте extend и include, а также простое обобщение, когда вам требуется выполнить декомпозицию вариантов использования на повторно использ уемые части.

Интерфейсы обсуждаются в главе 16.

Один из стереотипов вы будете встречать в контексте взаимодействия объектов:

- send – показывает, что исходный класс посылает сообщение целевому.

Машины состояний (конечные автоматы) обсуждаются в главе 22.

Используйте send, когда вам предстоит смоделировать операцию (такую как действие, связанное с переходом между состояниями), передающую данное событие целевому объекту, который, в свою очередь, может иметь ассоциированный конечный автомат. Зависимость send позволяет вам эффективно связать друг с другом два независимых автомата.

Системы и модели обсуждаются в главе 32.

И наконец, еще один стереотип вы встретите в контексте организации элементов вашей системы в подсистемы и модели:

- trace – показывает, что целевой элемент является историческим предшественником исходного (иначе говоря, тем же элементом, но на ранней стадии разработки).

Пять представлений архитектуры обсуждаются в главе 2.

Используйте trace, когда нужно смоделировать связи между элементами различных моделей. Например, в контексте архитектуры системы вариант использования в модели вариантов использования, представляющий функциональное требование, может трассироваться в пакет соответствующей модели дизайна, представляющей артефакты, которые реализуют данный вариант использования.

На заметку. Все связи, включая обобщение, ассоциацию и реализацию, являются концептуальными разновидностями зависимостей. Обобщение, ассоциация и реализация обладают довольно важной семантикой, достаточной для того, чтобы трактовать их как различные виды связей в UML. Перечисленные выше стереотипы отражают тонкие

нюансы зависимостей, каждая из которых наделена своей собственной семантикой, но не настолько семантически удалена от обычной зависимости, чтобы представлять ее как особый вид связи. В каком-то смысле это не очень отвечает идеологии UML, но опыт показывает, что подобный подход позволяет достичь баланса в определении важных видов связей, с которыми вы можете столкнуться и избежанием избыточного числа вариантов выбора для разработчика модели. Вы не ошибетесь, если сначала смоделируете обобщения, ассоциации и реализации, а все остальные виды связей обозначите зависимостями.

Обобщения

Основные свойства обобщений обсуждаются в главе 5.

Обобщение – это связь общего классификатора, называемого суперклассом или родителем, и более конкретного, называемого подклассом или дочерним классом. Например, у вас могут быть общий класс Window (Окно) и более конкретный – MultiPanelWindow (МногофорточноеОкно). Если установлена связь обобщения от дочернего к родительскому классу, то первый (MultiPanelWindow) унаследует всю структуру и поведение второго (Window). Притом дочерний класс может в дополнение к этому представить новую структуру и поведение или даже переопределить поведение родителя. В связи обобщения экземпляры потомка могут быть использованы везде, где применимы экземпляры родителя – это означает, что экземпляр потомка может быть подставлен вместо экземпляра родителя.

В большинстве случаев вам будет достаточно *одиночного наследования*. О классе, который имеет только одного родителя, говорят, что он использует одиночное наследование. Однако бывают случаи, когда один класс включает в себя аспекты множества других – тогда лучше смоделировать эти связи *множественным наследованием*. В качестве примера рассмотрим рис. 10.2, где показан набор классов, составляющих приложение финансовых услуг. У класса Asset (Активы) три потомка: BankAccount (БанковскийСчет), RealEstate (Недвижимость) и Security (ЦенныеБумаги). Два из них – BankAccount и Security – имеют своих собственных потомков. Так, Stock (Акция) и Bond (Облигация) – дочерние классы Security.

Дочерние классы BankAccount и RealEstate наследуют от множества родителей. Например, RealEstate – это разновидность как Asset, так и InsurableItem (ОбъектСтрахования), а BankAccount – разновидность Asset, InterestBearingItem (ОбъектНачисленияПроцентов), и InsurableItem.

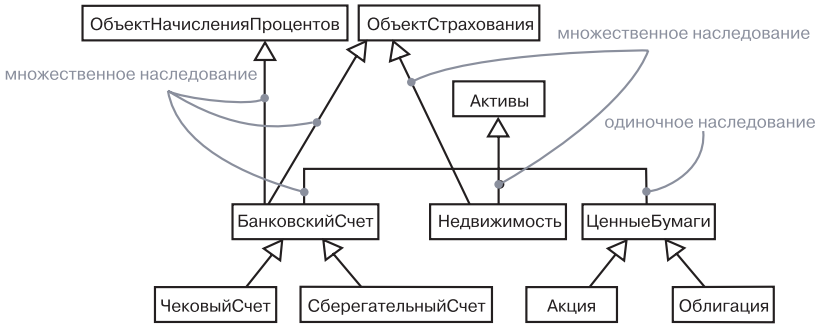


Рис. 10.2. Множественное наследование

Некоторые суперклассы используются только для того, чтобы добавить, как правило, поведение и иногда – структуру к классам, которые наследуют свою основную структуру от обычных суперклассов. Эти дополнительные классы называются *примесями* (mixins); они не могут применяться автономно, всегда выступая в качестве добавочных суперклассов в связи множественного наследования. Например, `InterestBearingItem` и `InsurableItem` на рис. 10.2 – примеси.

На заметку. Применяйте множественное наследование осторожно: вы можете столкнуться с проблемами, если дочерний класс имеет нескольких родителей, структура и поведение которых перекрываются. Во многих случаях множественное наследование может быть заменено *делегацией*, когда дочерний класс наследует только от одного родителя, а затем посредством агрегации перенимает структуру и поведение второстепенных родителей. Например, вместо специализации класса `Vehicle` (Транспорт), с одной стороны, по принципу `LandVehicle` (НаземныйТранспорт), `WaterVehicle` (ВодныйТранспорт) и `AirVehicle` (ВоздушныйТранспорт), а с другой, по принципу `GasPowered` (НаГазовойЭнергии), `WindPowered` (НаВетровойЭнергии) и `MusclePowered` (НаМускульнойЭнергии), допустите, чтобы класс `Vehicle` содержал в виде составной части `meansOfPropulsion` (движущаяСила). Главный недостаток данного подхода в том, что для таких второстепенных родителей утрачивается семантика подстановки.

Механизмы расширения UML обсуждаются в главе 6.

Простой, без дополнений, связи обобщения будет достаточно в большинстве случаев наследования, с которыми вы столкнетесь. Однако если вы хотите подчеркнуть тонкие смысловые оттенки, на помощь приходят существующие в UML ограничения, которые могут быть наложены на связи обобщения:

1. `complete` – показывает, что все д очерние классы в обобщении специфицированы в модели (хотя некоторые могут быть не указаны на диаграмме) и никакие дополнительные дочерние классы не допускаются;
2. `incomplete` – показывает, что не все дочерние классы в обобщении специфицированы в модели (даже если некоторые и пропущены) и допускается создание дополнительных дочерних классов.

Общие свойства диаграмм обсуждаются в главе 7.

Если только не указано иное, вы можете предполагать, что любая диаграмма показывает только частичное представление структуры наследования – остальная часть пропущена. Однако это не имеет отношения к завершенности модели. В частности, ограничение `complete` следует использовать, когда вы хотите явно показать, что иерархия наследования в модели показана полностью (хотя вполне возможно, что ни одна диаграмма не отражает всю иерархию целиком). Ограничение `incomplete` позволяет явно указать, что вы не устанавливаете полную иерархию в модели (хотя одна диаграмма может показать все, что в этой модели есть).

3. `disjoint` – означает, что объект родительского класса может принадлежать только одному типу классов-потомков. Например, класс `Person` (Человек) может быть специализирован путем создания `disjoint`-классов `Man` (Мужчина) и `Woman` (Женщина).
4. `overlapping` – означает, что объект родительского класса может принадлежать нескольким типам классов-потомков. И подклассы именно перекрывающиеся. Например, класс `Vehicle` (Транспорт) может быть специализирован путем создания подклассов `LandVehicle` (НаземныйТранспорт) и `WaterVehicle` (ВодныйТранспорт); автомобиль-амфибия относится к обоим.

Типы и интерфейсы обсуждаются в главе 11, взаимодействия – в главе 16.

Последние два ограничения относятся только к ситуациям множественного наследования. Используйте `disjoint`, чтобы показать, что классы в наборе являются взаимно несовместимыми; подкласс не может наследовать от нескольких родителей. Применяйте `overlapping`, если нужно показать, что класс допускает множественное наследование от более чем одного класса в наборе.

На заметку. В большинстве случаев во время исполнения объект имеет один тип – тогда мы имеем дело со *статической классификацией*. Если же объект может изменять свой тип во время исполнения, это пример *динамической классификации*. Моделировать последнюю сложно, но в UML вы можете вместо динамической классификации использовать сочетание множественного наследования (чтобы показать возможные типы объекта), типов и взаимодействий (чтобы показать изменение типа объекта во время исполнения).

Ассоциации

Основные свойства ассоциаций обсуждаются в главе 5.

Ассоциация – это структурная связь, показывающая, что объекты одной сущности соединены с объектами другой. Например, класс `Library` (Библиотека) может иметь ассоциацию «один-ко-многим» с классом `Book` (Книга), сообщая таким образом, что каждый экземпляр `Book` принадлежит одному экземпляру `Library`. Более того, вы можете найти библиотеку, где содержится конкретная книга, а в рамках одной библиотеки можете осуществить навигацию по всем ее книгам. Ассоциация изображается сплошной линией, соединяющей разные классы или же один – сам с собой. Используется, когда необходимо показать структурные связи.

Существуют четыре базовых дополнения, применимых к ассоциациям: имя, роль на каждом конце ассоциации, множественность каждого конца ассоциации и агрегация. Для расширенного применения предусмотрен ряд других свойств, которые вы можете использовать при моделировании тонких деталей: навигация, квалификация и разные типы агрегаций.

Навигация (navigation). Имея простую, без дополнений, ассоциацию между двумя классами (наподобие `Book` и `Library` в вышерассмотренном примере), вы можете осуществлять навигацию от объектов одного вида к объектам другого. Если только не указано иное, навигация по ассоциации двунаправлена. Иногда требуется ограничить ее лишь одним направлением. Например, при моделировании служб операционной системы (см. рис. 10.3) вы столкнетесь с ассоциацией между объектами `User` (Пользователь) и `Password` (Пароль). Для данного объекта `User` понадобится искать соответствующий ему объект `Password`, но по объекту `Password` нет смысла находить соответствующий объект `User`. Вы можете явно задать направление навигации, снабдив ассоциацию дополнением в виде стрелки, указывающей в нужную сторону.

На заметку. Указание направления обхода не обязательно означает, что вы не можете попасть от объекта, находящегося на одном конце ассоциации, к объекту на другом ее конце. Скорее, навигация констатирует «знание» одного объекта о другом. Так, в предыдущем примере можно найти объект `User`, ассоциированный с конкретным объектом `Password`, хотя ассоциации с другими классами не показаны. Свойство «навигабельности» ассоциации говорит о том, что, имея объект на одном ее конце, вы можете легко и напрямую получить объекты на другом, – обычно потому, что исходный объект сохраняет некую ссылку на целевые.

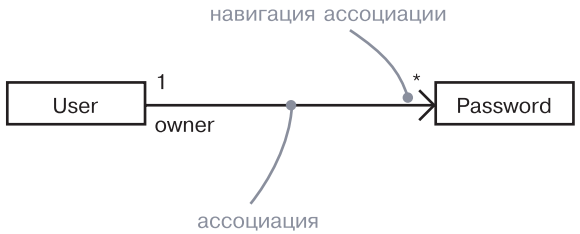


Рис. 10.3. Навигация

Открытая, защищенная, закрытая и пакетная видимость обсуждается в главе 9.

Видимость (visibility). При наличии ассоциации между двумя классами объекты одного класса могут «видеть» объекты другого и допускать навигацию к ним, если только она не ограничена явным образом. Однако иногда необходимо ограничить видимость одних объектов по отношению к другим в пределах ассоциации. В примере на рис. 10.4 ассоциации установлены между `UserGroup` (Группа-Пользователей) и `User` (Пользователь), а также между `User` и `Password` (Пароль). Имея объект `User`, можно идентифицировать соответствующие ему объекты `Password`. Однако `Password` является закрытым по отношению к `User` и не может быть доступен извне (конечно, если только пользователь не открывает доступ к паролю, – возможно, посредством какой-то открытой операции). Таким образом, по рисунку видно, что, имея объект `UserGroup`, вы можете перейти к его объектам `User` и наоборот, но не можете увидеть объекты `Password`, принадлежащие `User`, потому что они закрыты по отношению к `User`.

UML предусматривает три уровня видимости для конца ассоциации, назначаемые подобно тому, как вы делаете это в отношении свойств класса, добавляя символ видимости к имени роли. Если только не указано иное, видимость роли – открытая. Закрытая видимость означает, что объекты на данном конце ассоциации не доступны никаким объектам вне ее; защищенная видимость означает, что объекты на данном конце ассоциации не доступны никаким объектам вне ее, за исключением дочерних по отношению к тому, который находится на другом конце. Пакетная видимость означает, что классы, объявленные в том же пакете, могут видеть данный элемент; это не касается концов ассоциации.

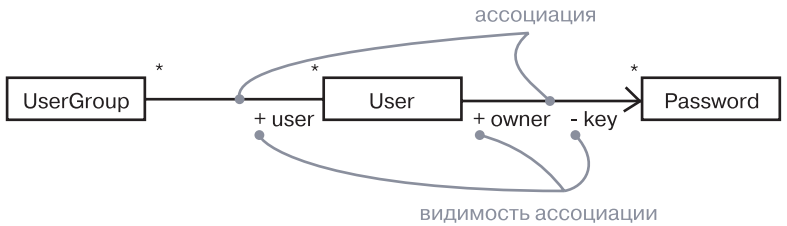


Рис. 10.4. Видимость

Атрибуты обсуждаются в главах 4 и 9.

Квалификация (qualification). Применительно к ассоциациям одна из наиболее часто используемых идиом моделирования, с которыми вы будете сталкиваться, – *проблема поиска* (lookup). Как, имея объект на одном конце ассоциации, идентифицировать один или множество объектов на другом ее конце? Рассмотрим в качестве примера моделирование рабочего стола в мастерской, на котором сортируются бракованные изделия. Как показано на рис. 10.5, вы должны смоделировать ассоциацию между двумя классами: `WorkDesk` (РабочийСтол) и `ReturnedItem` (ВозвращенноеИзделие). Для `WorkDesk` вам понадобится `jobId` – идентификатор определенного `ReturnedItem`. В этом смысле `jobId` – атрибут ассоциации. Он не является свойством `ReturnedItem`, поскольку сами изделия не несут в себе представления о ремонте. Поэтому, имея в наличии объект `WorkDesk` и определенное значение `jobId`, вы можете осуществить навигацию к нулю или одному объекту `ReturnedItem`. В UML данную идиому следует моделировать с помощью *квалификаторов* (qualifiers), которые являются атрибутами ассоциаций, значение которых идентифицирует подмножество объектов (чаще – один объект), связанных с другим объектом по ассоциации. Квалификатор изображается в виде маленького прямоугольника, присоединенного к концу ассоциации; в этот прямоугольник помещены атрибуты, как показано на рис. 10.5. Исходный объект вместе со значениями атрибутов квалификатора порождает целевой объект, если множественность целевого объекта равна единице, или же набор целевых объектов, если множественность более единицы.

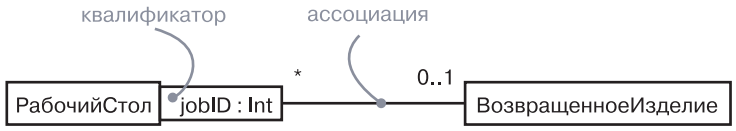


Рис. 10.5. Квалификация

Простая агрегация обсуждается в главе 5.

Композиция (composition). Агрегация представляет собой простую концепцию с довольно глубокой семантикой. Простая агрегация исключительно концептуальна и не заявляет ничего помимо отличия целого от части. Она не изменяет смысла навигации по ассоциации между целым и его частями и ничего не говорит о жизненном цикле связи целого с его частями.

Атрибуты – по существу сокращенное выражение композиции; атрибуты обсуждаются в главах 4 и 9.

Однако существует особая разновидность простой агрегации – композиция, которая добавляет ко всему вышеназванному некоторую важную семантику. *Композиция* – это форма агрегации с четко выраженными отношениями владения и совпадением времени жизни частей и целого. Части с нефиксированной множественностью могут быть созданы после самого композита, но однажды созданные, живут и умирают вместе с ним. Кроме того, такие части могут быть явно удалены перед «смертью» композита.

Это означает, что в композитной агрегации объект может быть одновременно частью только одного композита. Например, при моделировании оконной системы объект `Frame` (Рама) принадлежит только одному объекту – `Window` (Окно). Этот вариант отличается от простой агрегации, допускающей принадлежность части нескольким целым. Так, в модели дома `Wall` (Стена) может быть частью одного или нескольких объектов `Room` (Комната).

Вдобавок в композитной агрегации целое распоряжается своими частями; это значит, что композит должен управлять созданием и удалением своих частей. Например, когда вы создаете объект `Frame` в оконной системе, то должны присоединить его к включающему объекту `Window`. Аналогичным образом, когда объект `Window` уничтожается, он обязан ликвидировать свои части `Frame`.

Из рис. 10.6 видно, что композиция – это всего лишь особый вид ассоциации. Поэтому графически она представлена как простая ассоциация, которую дополняет закрашенный ромбик со стороны целого.

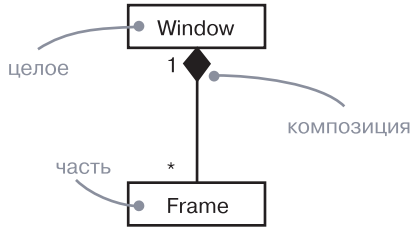


Рис. 10.6. Композиция

Внутренняя структура класса обсуждается в главе 15.

На заметку. Альтернативное изображение композиции предполагает использование структурированного класса и вложение символов частей в символы композита. Эта форма более удобна, когда требуется выразить связи между частями, существующими только в контексте целого.

Атрибуты обсуждаются в главах 4 и 9.

Ассоциации-классы (association class). Ассоциация, связывающая два класса, сама по себе может обладать некоторыми свойствами. Например, на рис. 10.7 в связи «работодатель/работник» между классами `Company` (Компания) и `Person` (Человек) присутствует элемент `Job` (Должность) – свойство ассоциации, описывающее должность работника в компании и соединяющее каждую пару объектов `Company` и `Person`. Было бы неверно моделировать эту ситуацию ассоциациями `Company` с `Job` и `Job` с `Person`, поскольку при этом не определяется конкретный экземпляр `Job`, связывающий друг с другом классы `Company` и `Person`.

В UML такой случай описывается ассоциацией-классом – элементом, который одновременно имеет свойства класса и ассоциации. Его можно рассматривать двояко: как ассоциацию, которая несет в себе свойства класса, или же как класс, наделенный свойствами ассоциации. Изображается он символом класса, соединенным пунктирной линией с линией ассоциации (см. рис. 10.7).

На заметку. Иногда необходимо, чтобы одни и те же свойства присутствовали в нескольких ассоциациях-классах. Однако вы не можете присоединить ассоциацию-класс к нескольким ассоциациям, поскольку она, по сути, тоже является ассоциацией. Чтобы добиться желаемого, определите обычный класс (скажем, под названием `C`), обладающий этим свойством, и сделайте его потомками все ассоциации-классы, которым это свойство понадобится, или же используйте `C` в качестве типа атрибута.

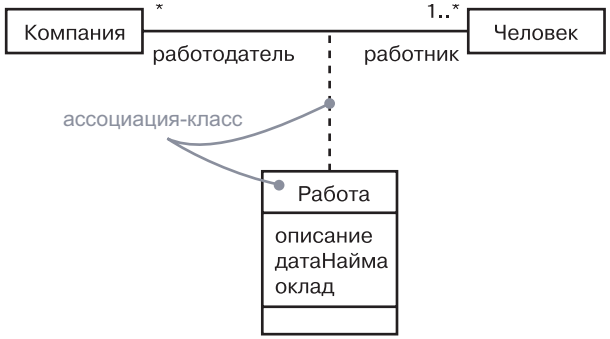


Рис. 10.7. Ассоциации-классы

Ограничения. Эти простые и расширенные свойства ассоциаций понадобятся в большинстве случаев использования структурных связей. Однако для выражения тонких смысловых оттенков UML определяет пять ограничений, которые могут касаться связей ассоциации.

Механизмы расширения UML обсуждаются в главе 6.

Во-первых, вы можете показать, что объекты на одном конце ассоциации (с множественностью больше единицы) упорядочены или не упорядочены:

1. `ordered` – сообщает, что набор объектов на конце ассоциации должен быть явно упорядочен.

Например, в ассоциации `User/Password` объекты `Password`, ассоциированные с `User`, могут сохраняться в обратном порядке (то есть первым идет последний использованный пароль), и, соответственно, ассоциация может быть помечена как `ordered`. Если это ключевое слово отсутствует, то объекты не упорядочены.

Свойства изменяемости также касаются атрибутов, как это описано в главе 9; связи обсуждаются в главе 16.

Во-вторых, вы можете подчеркнуть, что объекты на одном конце ассоциации должны быть уникальными (то есть формировать набор) или же не должны быть таковыми (то есть формировать множество с повторяющимися элементами – bag):

2. `set` – объекты уникальны, без дубликатов;
3. `bag` – объекты не уникальны, допускаются дубликаты;
4. `ordered set` – объекты уникальны и упорядочены;
5. `list` или `sequence` – объекты упорядочены, могут быть дубликаты.

И наконец, есть ограничение изменяемости экземпляров ассоциации:

6. `readonly` – ссылка, однажды установленная от объекта на противоположном конце ассоциации, не может быть модифицирована или удалена. По умолчанию при отсутствии этого ограничения допускается изменяемость ассоциации.

На заметку. По сути, `ordered` и `readonly` – это свойства конца ассоциации. Однако они изображаются в нотации ограничений.

Реализация

Реализация – это семантическая связь между классификаторами, один из которых специфицирует некий контракт, а другой обязуется его выполнять. Изображается пунктирной линией с большой незакрашенной стрелкой, указывающей на классификатор, который специфицирует контракт.

Интерфейсы обсуждаются в главе 11, классы – в главах 4 и 9, компоненты – в главе 15, пять представлений архитектуры – в главе 2.

Реализация существенно отличается от связей зависимости, обобщения и ассоциации, а потому трактуется как отдельный тип связи. Семантически она являет собой нечто среднее между зависимостью и обобщением, и ее нотация представляет собой сочетание нотаций зависимости и обобщения. Реализацию используют в двух случаях: в контексте интерфейсов и в контексте кооперации.

В основном вы будете применять реализацию для описания связи между интерфейсом и классом (или же компонентом, который предоставляет операцию либо сервис для него). *Интерфейс* – это набор операций, используемых для спецификации сервиса класса или компонента. Таким образом, интерфейс описывает контракт, который класс или компонент обязаны исполнять. Интерфейс может быть реализован многими такими классами или компонентами, и напротив, класс или компонент может реализовывать множество интерфейсов.

Возможно, самое интересное в интерфейсах то, что они позволяют вам отделить спецификацию контракта (то есть сам интерфейс) от его реализации классом или компонентом. Более того, интерфейсы связывают логическую и физическую части архитектуры системы. Например, как показано на рис. 10.8, класс `AccountBusinessRules` (БизнесПравилаБанковскогоСчета) в системе ввода заказов в представлении дизайна системы может реализовывать такой интерфейс, как `IruleAgent` (IАгентПравила). Тот же самый интерфейс может быть реализован компонентом (в данном случае – `acctrule.dll`) в представлении реализации системы.

Отметим, что вы можете представить реализацию двумя способами: в канонической форме (применяя стереотип `interface` и пунктирную линию с большой незакрашенной стрелкой) или в сокращенной форме (используя нотацию «леденца» (lollipop notation) для предоставляемого интерфейса).

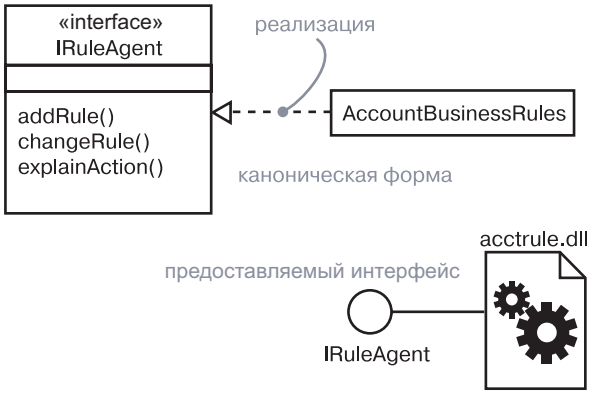


Рис. 10.8. Реализация интерфейса

Варианты использования обсуждаются в главе 17.

Реализацию следует применять также для изображения связи между вариантом использования и кооперацией, которая его реализует (см. рис. 10.9). В этом случае почти всегда рисуется пунктирная линия со стрелкой.

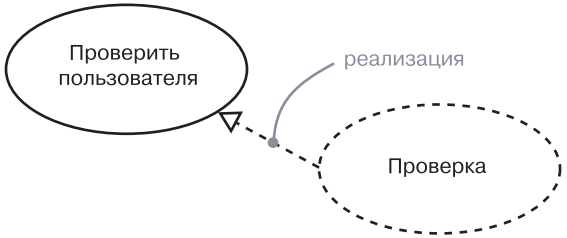


Рис. 10.9. Реализация варианта использования

На заметку. Когда класс или компонент реализует интерфейс, это означает, что клиенты могут полагаться на данный класс/компонент в том, что он позаботится о поведении, специфицированном в интерфейсе. То есть класс или компонент реализует все операции интерфейса, будет отвечать на все его сигналы и во всем следовать протоколу, установленному интерфейсом для клиентов, которые используют эти операции или посылают эти сигналы.

Типичные приемы моделирования

Моделирование систем связей

Моделирование словаря системы и распределения обязанностей в системе обсуждается в главе 4.

При моделировании словаря сложной системы вы можете столкнуться с дюжинами, если не с сотнями и тысячами классов, интерфейсов, компонентов, узлов и вариантов использования. Трудно установить четкие границы между всеми этими абстракциями. Еще сложнее описать мириады связей между ними. Это потребует от вас сбалансированного распределения обязанностей в системе в целом, с применением индивидуальных абстракций, которые обладают высокой согласованностью, выразительностью и слабой связностью.

Когда вы моделируете эти системы связей, пользуйтесь следующими рекомендациями:

Варианты использования обсуждаются в главе 17.

- не начинайте этот процесс в отрыве от остальных. Применяйте варианты использования и сценарии для исследования связей среди множества абстракций;
- вообще, начинайте моделирование с реально существующих структурных связей. Это дает статическое представление о системе, в достаточной мере ее проясняя;

Пять представлений архитектуры обсуждаются в главе 2. Технология Rational Unified Process описывается в приложении 2.

- далее идентифицируйте возможности связей обобщения/специализации. Осторожно применяйте множественное наследование;
- только по завершении предыдущих действий обратите внимание на зависимости. Обычно они представляют более тонкую форму семантических связей.
- описывая каждый вид связей, начинайте с его базовой формы и добавляйте расширенные средства, только если без них не обойтись для выражения ваших намерений;
- помните, что не обязательно и даже нежелательно демонстрировать все существующие связи в наборе абстракций на единственной диаграмме или в одном представлении. Вместо этого выстройте связи внутри проектируемой системы, показывая

их в разных ее представлениях. Наборы связей, которые могут вызвать особый интерес, выносите в отдельные диаграммы.

Ключ к успешному моделированию сложных систем связей лежит в пошаговом подходе. Занимайтесь построением связей по мере того, как усложняете структуру системной архитектуры. Упрощайте эти отношения, исследуя альтернативы общепринятым механизмам. В процессе разработки оценивайте связи между ключевыми абстракциями вашей системы в каждой версии.

На заметку. На практике (особенно если вы следуете правилам пошагового и итерационного процесса разработки) связи в ваших моделях будут зависеть от решений, принятых разработчиком моделей, а также от обратного проектирования вашей реализации.

Советы и подсказки

Когда вы моделируете расширенные связи в UML, помните о том, что вам предоставлен широкий выбор строительных блоков, – от простых ассоциаций до более детализированных свойств навигации, квалификации, агрегации и т.п. Вы должны выбирать такие связи и их детали, которые наилучшим образом подходят для ваших абстракций. Хорошо структурированная связь:

- раскрывает только те средства, которые необходимо использовать клиентам, и скрывает остальные;
- недвусмысленна по назначению и семантике;
- не настолько конкретизирована, чтобы ограничить степень свободы в ее реализации;
- не настолько абстрактна, чтобы допустить неоднозначное толкование.

Когда вы изображаете связь в UML:

- показывайте только те свойства связи, которые важны для понимания абстракции в ее контексте; выбирайте версию со стереотипом, наилучшим образом выражающую назначение связи визуально.

Глава 11. Интерфейсы, типы и роли

В этой главе:

- **Интерфейсы, типы, роли и реализация**
- **Моделирование соединений в системе**
- **Моделирование статических и динамических типов**
- **Обеспечение понятности и доступности интерфейсов**

Интерфейсы очерчивают границу между выражением того, что делает абстракция, и реализацией того, как она это делает. Интерфейс – это набор операций, применяемый для описания сервиса класса или компонента.

Интерфейсы используются для визуализации, специфицирования, конструирования и документирования соединений внутри системы. Типы и роли представляют механизм моделирования статического и динамического согласования абстракции с интерфейсом в определенном контексте.

Хорошо структурированный интерфейс представляет четкое разделение внешнего и внутреннего представлений абстракции, обеспечивая понятность и доступность без необходимости погружаться в детали его реализации.

Введение

Проектирование на примере строительства дома обсуждается в главе 1.

Не слишком разумно проектировать дом так, чтобы всякий раз, когда вы захотите перекрасить стены, потребовалось бы разрушать фундамент. И вряд ли вы захотите жить в доме, где для замены светильника пришлось бы менять всю электропроводку. Домовладелец не придет в восторг от необходимости замены всех электрических и телефонных розеток при каждой смене квартиросъемщика.

Столетия строительного опыта позволили выработать множество прагматичных конструктивных приемов, позволяющих строителям избежать этих – иногда очевидных, иногда не слишком очевидных – проблем, возникающих в процессе строительства и ремонта. В терминах программного обеспечения мы назовем это дизайном с четким разделением составляющих компонентов. Например,

в хорошо структурированном доме внешний вид фасада всей конструкции можно изменить или заменить, не заботясь об остальной части дома. Аналогичным образом смена меблировки не должна глобально влиять на интерьер. Коммуникации, проложенные в стенах для электро-, тепло- и водоснабжения, а также канализации, не получится провести заново без особых трудозатрат, но все же вам не придется обращаться с этой целью в домостроительный комбинат.

Но не только стандартная практика строительства помогает строить дома, поддающиеся перепланировке. Этому способствует и множество стандартных интерфейсов строительства, позволяющих использовать общие готовые компоненты, применение которых существенно снижает затраты на строительство и эксплуатацию. Например, деревянные строительные конструкции имеют стандартные размеры, что позволяет строить стены, длина и ширина которых кратна известному числу. Предусмотрены стандартные размеры дверей и окон, обеспечивающие относительную легкость их открытия. Наконец, существуют стандарты на электрические и телефонные розетки (хотя они и отличаются в разных странах), что позволит вам легко переставить электроприборы в случае необходимости.

Каркасы (frameworks) обсуждаются в главе 29.

В программном обеспечении важно строить системы с четким разделением аспектов, чтобы по мере их эволюции изменения в одной части системы не затрагивали и не повреждали остальные ее части. Один из главных способов достижения этой цели – указание четких соединений (швов) между частями, которые могут изменяться независимо друг от друга. Более того, применяя правильные интерфейсы, вы можете вместо того, чтобы строить их самостоятельно, выбирать для их реализации стандартные компоненты, библиотеки и каркасы. Если вы найдете более удачные реализации, можно будет заменить ими старые, не вводя в недоумение пользователей.

Классы обсуждаются в главах 4 и 9, компоненты – в главе 15. Пакеты обсуждаютсяв главе 12, подсистемы – в главе 32. Компоненты обсуждаются в главе 15.

В UML интерфейсы используются для моделирования соединений в системе. Объявляя интерфейс, вы можете определить необходимое поведение абстракции независимо от ее реализации. Клиенты могут строить вокруг этих интерфейсов все, что им нужно, и вы вправе строить или покупать любые реализации данного интерфейса до тех пор, пока они удовлетворяют обязанностям и контракту, обозначенному интерфейсом.

Многие языки программирования, включая Java и CORBA IDL, поддерживают концепцию интерфейса. Интерфейсы – это важное средство не только разделения спецификации и реализации класса или компонента, но и определения внешнего представления пакета или подсистемы.

Интерфейсы в UML изображаются так, как показано на рис. 11.1. Такая нотация позволяет раздельно визуализировать описание абстракции и ее реализацию.

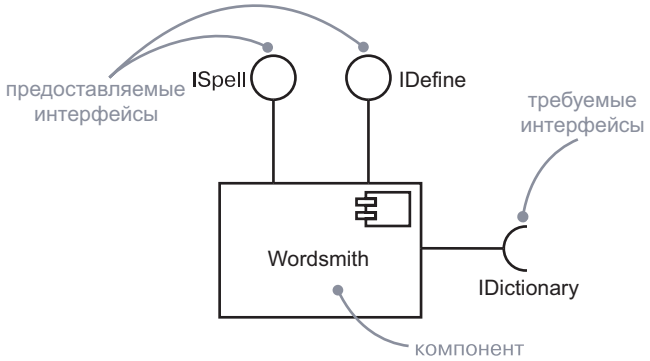


Рис. 11.1. Интерфейсы

Базовые понятия

Интерфейс (interface) – это набор операций, используемый для описания сервиса класса или компонента. *Тип* (type) – это стереотип класса, используемый для спецификации домена объектов вместе с разрешенными для них операциями (но не методами). *Роль* (role) – это поведение сущности в определенном контексте.

Графически интерфейс может быть представлен как класс со стереотипом – это позволяет раскрыть его операции и прочие свойства. Для отображения связи между классом и интерфейсом используется специальная нотация. Предоставляемый интерфейс – тот, который описывает сервисы, обеспечиваемые классом, – показан в виде маленького кружка, присоединенного к прямоугольнику (классу). Требуемый интерфейс – тот, который один класс требует от другого, – выглядит как маленький полукруг, соединенный с прямоугольником (классом).

На заметку. Помимо прочего интерфейсы могут использоваться в целях описания контракта для варианта использования или подсистемы.

Имена

Имена интерфейсов должны быть уникальны в пределах включающего их пакета (см. главу 12).

У каждого интерфейса должно быть имя, отличное от имен других интерфейсов. *Имя* – это текстовая строка. Само по себе оно рассматривается как *простое имя*; *квалифицированное имя* – это имя интерфейса, предваренное именем пакета, в котором он содержится. Интерфейс может быть нарисован с указанием только имени (см. рис. 11.2).

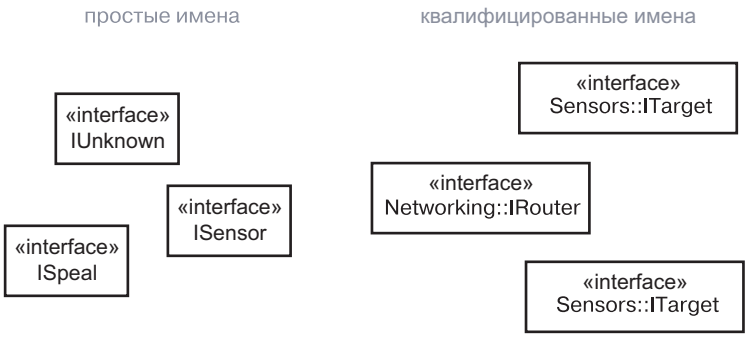


Рис. 11.2. Простые и квалифицированные имена

На заметку. В имени интерфейса могут в любом количестве использоваться буквы латинского алфавита, цифры и некоторые знаки пунктуации (за исключением таких, как двоеточие, которое применяется для отделения имени интерфейса от имени содержащего его пакета). Имя может располагаться в несколько строк. Фактически имена интерфейсов – это краткие существительные или фразы-существительные, взятые из словаря моделируемой системы.

Операции

Операции описываются в главах 4 и 9, механизмы расширения UML – в главе 6.

Интерфейс – это именованный набор операций, применяемый для описания сервиса класса или компонента. В отличие от классов или типов, интерфейсы не описывают никакой реализации (то есть не могут включать в себя никаких методов, представляющих реализации операций). Подобно классу, интерфейс может содержать любое количество операций. Они могут быть дополнены свойствами видимости, параллельности, стереотипами, помеченными значениями и ограничениями.

Когда вы декларируете интерфейс, то изображаете его как класс со стереотипом, перечисляя операции в соответствующем разделе. Операции могут быть представлены только именем либо полной сигнатурой и прочими свойствами (см. рис. 11.3).

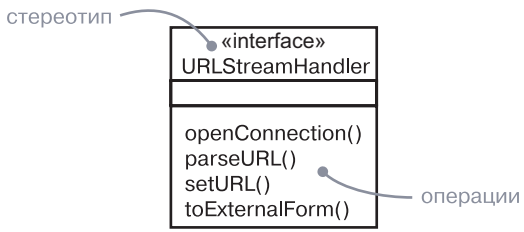


Рис. 11.3. Операции

События обсуждаются в главе 21.

На заметку. С интерфейсами также можно ассоциировать сигналы.

Связи

Связи обсуждаются в главах 5 и 10.

Подобно классу, интерфейс может участвовать в связях обобщения, ассоциации и зависимости, а кроме того, в связи реализации. Реализация – это семантическая связь между двумя классификаторами, один из которых описывает контракт, а другой обязуется его исполнять.

Интерфейс специфицирует контракт для класса или компонента, не навязывая его реализации. Класс или компонент может реализовывать множество интерфейсов – в таком случае он обязуется исполнять все контракты точно и в полной мере, то есть предоставлять набор методов, которые правильно реализуют операции, определенные во всех этих интерфейсах. Набор предлагаемых сервисов называется *предоставляемым интерфейсом*. Аналогичным образом класс или компонент может зависеть от множества интерфейсов. При этом он ожидает, что контракты будут предоставлены некоторым набором компонентов, реализующих интерфейсы. Набор сервисов, которые данный класс требует от других классов, называется *требуемым интерфейсом*. Вот почему мы говорим, что интерфейс представляет собой соединительный элемент, связывающий компоненты системы. Интерфейс специфицирует контракт, две стороны которого – клиент и поставщик – могут изменяться независимо до тех пор, пока каждый из них соблюдает свои договорные обязанности.

Как явствует из рис. 11.4, вы можете двумя способами показать, что элемент реализует интерфейс. Во-первых, существует простая форма: интерфейс и его связь реализации изображаются в виде линии, соединяющей прямоугольник (класс) и маленький кружок (в случае использования предоставляемого интерфейса) или маленький полукруг (в случае использования требуемого интерфейса). Эта форма удобна и даже предпочтительна, когда вы просто хотите показать соединения в вашей системе. Однако ограничения данной нотации в том, что вы не можете визуализировать операции или сигналы, представленные интерфейсом. Во-вторых, можно использовать расширенную форму: интерфейс изображается в виде класса со стереотипом, что позволяет визуализировать операции и другие свойства, а также нарисовать связь реализации (для предоставляемого интерфейса) или зависимости (для требуемого интерфейса) от классификатора или компонента к интерфейсу. В UML

связь реализации изображается пунктирной линией с большой треугольной стрелкой на конце, указывающей на интерфейс. Эта нотация – нечто среднее между обобщением и зависимостью.

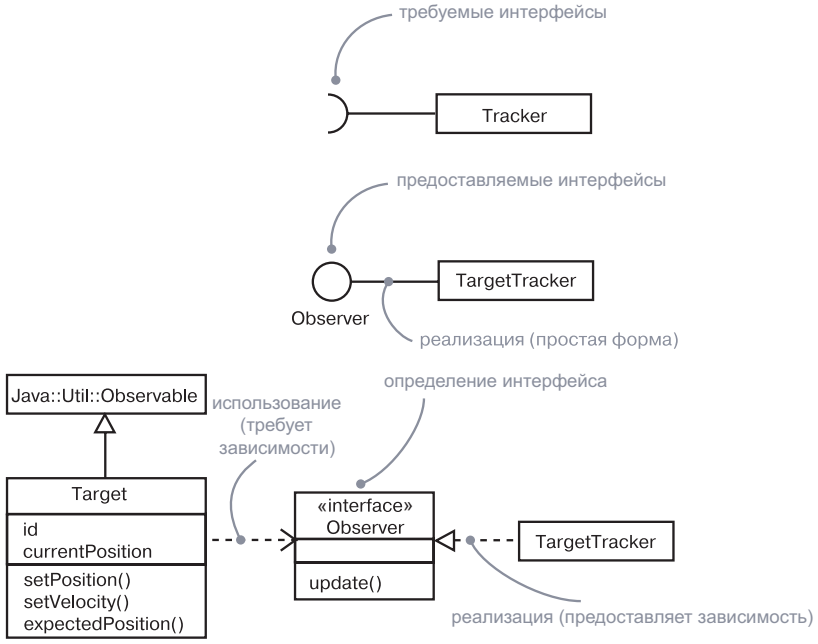


Рис. 11.4. Реализация

Абстрактные классы обсуждаются в главе 4, компоненты – в главе 15.

На заметку. Интерфейсы похожи на абстрактные классы. В частности, ни те, ни другие не могут иметь непосредственных экземпляров. Абстрактный класс, однако, может реализовывать свои конкретные операции. Интерфейс больше напоминает абстрактный класс, у которого абстрактны и все операции.

Понимание интерфейсов

Операции и их свойства обсуждаются в главе 9, семантика параллелизма рассматривается в главе 24.

Первое, что вы видите при работе с интерфейсом, – это набор операций, специфицирующих сервис класса или компонента. Если посмотреть немного глубже, можно выявить полные сигнатуры этих операций наряду с их особыми свойствами, такими как видимость, контекст и семантика параллелизма.

Эти свойства важны, но для сложных интерфейсов их недостаточно, чтобы прояснить семантику предоставляемого ими сервиса и дать понять, как правильно использовать операции. При отсутствии

любых других сведений вы должны погрузиться в некую абстракцию, которая реализует интерфейс, чтобы понять, что именно делает каждая операция и как все они должны работать вместе. Однако это лишает смысла использование интерфейса, назначение которого состоит в том, чтобы представлять четкое разделение аспектов в системе.

Предусловия, постусловия, инварианты обсуждаются в главе 9, автоматы – в главе 22, кооперации – в главе 28, OCL – в главе 6.

В модели UML можно указать значительно больше информации, чтобы сделать интерфейс понимаемым и доступным. Во-первых, вы можете сопроводить каждую операцию пред- и постусловиями, а класс или компонент в целом – инвариантами. В результате клиент, желающий использовать интерфейс, сможет понять, что он делает и как его применять, не погружаясь в детали реализации. Если важны формальности, стоит прибегнуть к OCL для спецификации семантики. Во-вторых, можно сопроводить интерфейс конечным автоматом для описания допустимой частичной упорядоченности его операций. В-третьих, сопроводив интерфейс кооперациями, вы определяете его ожидаемое поведение с помощью ряда диаграмм взаимодействия.

Типичные приемы моделирования

Моделирование соединений в системе

Компоненты обсуждаются в главе 15, системы – в главе 32.

Наиболее общее назначение интерфейсов – моделировать соединения в системах, состоящих из программных компонентов, как Eclipse, .NET или Java Beans. Некоторые компоненты вы будете заимствовать из других систем, другие – покупать, а иные создавать самостоятельно, «с нуля». В любом случае вам придется писать связующий код, предназначенный для их соединения. А это потребует понимания интерфейсов, предоставляемых и требуемых каждым компонентом.

Идентификация соединений в системе включает в себя идентификацию четких линий разделения архитектуры. С каждой стороны от этих линий вы найдете компоненты, которые могут изменяться независимо, не оказывая влияния на противоположный компонент (до тех пор пока компоненты на обеих сторонах придерживаются требований контракта, специфицированного интерфейсом).

Образцы и каркасы обсуждаются в главе 29.

Когда вы используете компонент из другой системы или покупаете его, то, вероятно, находите в нем ряд операций с каким-то минимумом документации, где описывается суть каждой из них. Это удобно, но, к сожалению, недостаточно. Гораздо важнее для вас понимать порядок, в котором следует вызывать операции, и механизм, который они заключают в себе. Имея малодокументированный

компонент, лучшее, что вы можете сделать, – путем проб и ошибок построить концептуальную модель работы его интерфейса. Затем вы вправе изложить свое понимание предмета, моделируя соединение в системе с помощью интерфейсов на UML, с тем чтобы впоследствии вы и другие пользователи без труда в нем разбирались. Точно так же, когда вы создаете свой собственный компонент, то должны обеспечить понимание контекста, в котором он должен применяться, – иными словами, специфицировать интерфейсы, на которые он опирается при выполнении своей работы, а также интерфейсы, которые он предоставляет окружающей среде.

На заметку. Большинство компонентных систем наподобие Eclipse, .NET и Enterprise Java Beans открыто для *компонентной интроспекции*, то есть вы можете программно опросить интерфейс, чтобы узнать его операции. Это первый шаг к пониманию природы любого малодокументированного компонента.

Чтобы смоделировать соединения в системе, выполните следующие действия:

- Проведите границы между теми классами и компонентами вашей системы, которые более тесно взаимодействуют друг с другом, чем другие наборы классов и компонентов.
- Проверьте правильность объединения элементов в группы, рассмотрев последствия возможных изменений. Классы или компоненты, которые изменяются совместно, объедините в кооперации.
- Рассмотрите операции и сигналы, которые пересекают границы, проведенные на этапе 1, от экземпляров из одного набора классов или компонентов к экземплярам из другого набора.
- Объедините логически связанные наборы операций и сигналов в интерфейсы.
- Для каждой кооперации в системе определите интерфейсы, которые ей требуются (импортируемые), и те, которые она предоставляет (экспортирует) другим кооперациям. Импорт интерфейсов моделируйте связями зависимости, а экспорт – связями реализации.
- Для каждого такого интерфейса документируйте его динамику, используя пред- и постусловия каждой операции, а также варианты использования и автоматы для интерфейса в целом.

Кооперации обсуждаются в главе 28.

Моделирование поведения обсуждается в частях IV и V.

В качестве примера рассмотрим рис. 11.5, где показаны соединения, окружающие компонент `Ledger` (Гроссбух), взятый из системы

управления финансами. Этот компонент предоставляет (реализует) три интерфейса: `Iunknown` (IНеизвестный), `ILedger` (IГроссбух) и `Ireports` (IОтчеты). На данной диаграмме `IUnknown` показан в своей расширенной форме, другие два интерфейса – в упрощенной, как «леденцы» (lollipops). Все три интерфейса и экспортированы для использования другими компонентами.

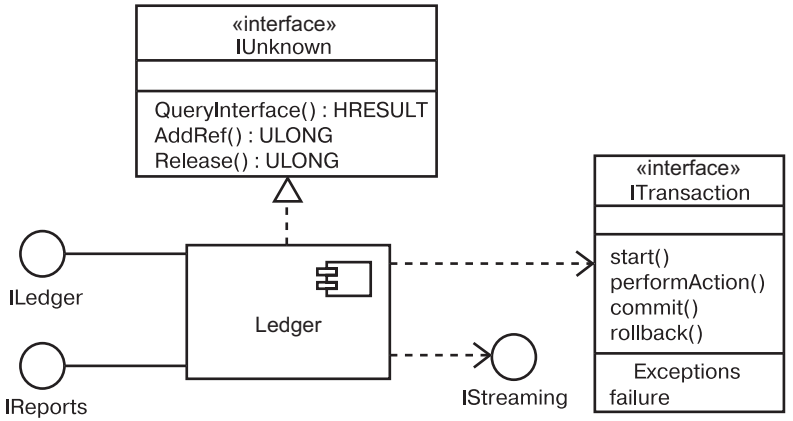


Рис. 11.5. Моделирование соединений в системе

Также на диаграмме показано, что `Ledger` требует (использует) два интерфейса: `IStreaming` (IПоток) и `Itransaction` (IТранзакция), причем второй показан в расширенной форме. Эти два интерфейса необходимы компоненту `Ledger` для правильной работы. Таким образом, в работающей системе вы должны подставить компоненты, которые реализуют оба этих интерфейса. Идентифицируя такие интерфейсы, как `ITransaction`, вы получаете эффективно разъединенные компоненты, лежащие по разные стороны интерфейса, что позволяет «нанимать» любой компонент, который соответствует интерфейсу.

Варианты использования обсуждаются в главе 17.

Такие интерфейсы, как `Itransaction`, представляют собой нечто большее, чем пул операций. Указанный интерфейс делает определенные предположения о порядке вызова его операций. Хотя это здесь и не продемонстрировано, вы должны сопроводить его вариантами использования и перечислить общие способы его применения.

Моделирование статических и динамических типов

Экземпляры обсуждаются в главе 13.

Большинство объектно-ориентированных языков программирования статически типизировано. Это означает, что с объектом в момент его создания связывается определенный тип (даже несмотря

на то, что объект, вероятно, будет впоследствии играть различные роли). Следовательно, клиенты, использующие объект, взаимодействуют с ним через разные наборы интерфейсов, которые представляют интересующие их множества операций (возможно, перекрывающиеся).

Диаграммы классов обсуждаются в главе 8.

Моделирование статической природы объекта может быть показано на диаграмме классов. Однако когда вы моделируете, скажем, бизнес-объекты, которые обычно изменяют свой тип в потоке работ, иногда целесообразно подчеркнуть динамическую природу их типа, задав ее явным образом. При таких обстоятельствах в течение своего жизненного цикла объект может принимать и терять типы. Смоделировать жизненный цикл объекта можно при помощи машины состояний (конечного автомата).

Чтобы смоделировать *динамический тип*, необходимо:

Ассоциации и обобщения обсуждаются в главах 5 и 10, диаграммы взаимодействий – в главе 19, зависимости – в главах 5 и 10.

- Специфицировать возможные типы данного объекта, представляя каждый из них в виде класса (если абстракция требует структуры и поведения) или же интерфейса (если абстракция требует только поведения).
- Смоделировать все роли, которые может играть класс объекта в любой момент времени. Вы можете пометить их стереотипом `<dynamic>` (он не предусмотрен в UML изначально – вы создаете его сами).
- На диаграмме взаимодействия правильно изобразить каждый экземпляр динамически типизированного класса. Отобразить тип экземпляра как состояние, – в фигурных скобках прямо под именем объекта. (Мы используем синтаксис UML новым способом, который, на наш взгляд, согласуется с выражением состояний.)

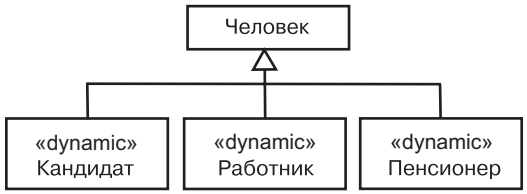


Рис. 11.6. Моделирование динамических типов

На рис. 11.6 показано, какие роли исполняют экземпляры класса `Person` (Человек) в системе управления персоналом.

По диаграмме видно, что экземпляры класса `Person` могут быть одного из следующих типов: `Candidate` (Кандидат), `Employee` (Работник) или `Retiree` (Пенсионер).

Советы и подсказки

Когда вы моделируете интерфейсы в UML, помните, что каждый из них должен представлять соединение в системе, отделяющее спецификацию от реализации. Хорошо структурированный интерфейс обладает следующими свойствами:

- прост, но полон – представляет все необходимые операции для описания отдельного сервиса;
- понятен – дает достаточно информации как для использования, так и для реализации, чтобы не нужно было исследовать имеющиеся случаи использования и реализации;
- доступен – представляет информацию пользователю по ключевым свойствам, без перегрузки деталями множества операций.

Изображая интерфейс в UML, учитывайте следующее:

- используйте нотацию «леденца» или сокета (socket) всякий раз, когда нужно показать наличие соединения в системе. В основном это понадобится для компонентов, а не для классов;
- используйте расширенную форму, когда необходимо представить детали самого сервиса. В большинстве случаев это понадобится для описания соединений между пакетами и подсистемами.

Глава 12. Пакеты

В этой главе:

- Пакеты, видимость, импорт и экспорт
- Моделирование групп элементов
- Моделирование архитектурных представлений
- Масштабирование больших систем

Визуализация, специфицирование, конструирование и документирование больших систем предполагают работу с множеством классов, интерфейсов, узлов, компонентов, диаграмм и других элементов. Масштабируя такие системы, вы столкнетесь с необходимостью организовывать эти сущности в крупные блоки. В языке UML для организации элементов модели в группы применяются пакеты.

Пакет – это способ организации элементов модели в блоки, которыми можно распоряжаться как единым целым. Можно управлять видимостью элементов пакета, так что некоторые будут видны пользователю, а другие – скрыты. Кроме того, с помощью пакетов изображаются различные представления архитектуры системы.

Хорошо структурированный пакет объединяет семантически близкие элементы, которые имеют тенденцию изменяться совместно. Такие пакеты характеризуются слабой связанностью и высокой согласованностью, причем доступ к содержимому пакета строго контролируется.

Введение

Разница в принципах постройки собачьей будки и небоскреба рассматривается в главе 1.

Устройство собачьей будки не представляет собой ничего сложного: четыре стены, лаз для собаки в одной из них и крыша. Для изготовления будки понадобится всего лишь несколько досок – конструкция не требует большего.

Жилые дома устроены сложнее. Стены, потолки и полы соединяются друг с другом, образуя более крупные сущности, которые мы называем комнатами. Комнаты организованы в еще более крупные блоки – жилая зона, зона досуга и т.д. Такие блоки могут представлять собой всего лишь абстракцию. Мы просто объединяем под общим названием ряд комнат, функционально связанных друг

с другом, чтобы удобнее было говорить о предполагаемом использовании домашнего пространства.

Небоскребы устроены еще сложнее. Там вы найдете помимо простейших конструкций (стен, полов, потолков) более крупные образования – места, открытые для общего доступа, сдаваемые в аренду квартиры и офисы. Эти блоки, вероятно, сгруппированы в еще более крупные, такие как арендуемая площадь и инфраструктура обслуживания здания. Разумеется, они не имеют ничего общего с самим зданием, а являются лишь артефактами, которые применяли при планировании небоскреба.

Каждая большая система состоит из нескольких подобных слоев. По-настоящему понять ее можно, только объединив абстракции, которые в нее входят, в крупные группы. Большинство блоков среднего размера (например, комнаты) сами по себе не являются абстракциями, похожими на классы, и у них бывает много экземпляров. Крупные блоки достаточно часто умозрительны (например, «часть здания для продажи»), и у них не бывает экземпляров. Они не реализуются на физическом уровне; единственная их цель – облегчить понимание системы. Блоки такого рода не будут представлены в развернутой системе – они существуют лишь на уровне модели.

Программная архитектура обсуждается в главе 2, моделирование архитектуры системы – в главе 32.

В языке UML организующие модель блоки называют *пакетами*. Пакет является универсальным механизмом организации элементов в группы, упрощающим понимание модели. Кроме того, пакеты позволяют контролировать доступ к своему содержимому, что облегчает работу с соединениями в архитектуре системы.

Графически пакеты в языке UML представлены так, как показано на рис. 12.1. Такая нотация позволяет визуализировать группы элементов, с которыми можно обращаться как с единым целым, контролируя при этом видимость и возможность доступа к отдельным элементам.

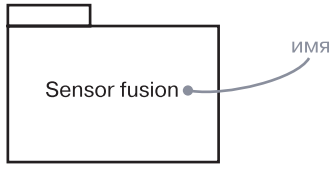


Рис. 12.1. Пакеты

Базовые понятия

Пакет (package) представляет собой общий механизм организации элементов в группы. Изображается в виде папки с закладкой. Имя пакета указано на папке, если содержимое последней не показано, а в противном случае – на закладке.

Имена

У каждого пакета должно быть имя, отличающее его от других пакетов. Имя представляет собой текстовую строку. Само по себе оно называется *простым*. *Квалифицированное* имя предваряется именем пакета, включающего данный, если такое вложение имеет место. Двойное двоеточие (::) используется в качестве разделителя имен пакетов. Обычно, изображая пакет, указывают только его имя (см. рис. 12.2). Но, как и в случае с классами, вы можете дополнять пакеты помеченными значениями или дополнительными разделами, чтобы прояснить детали.

Имя пакета должно быть уникальным внутри включающего его пакета.

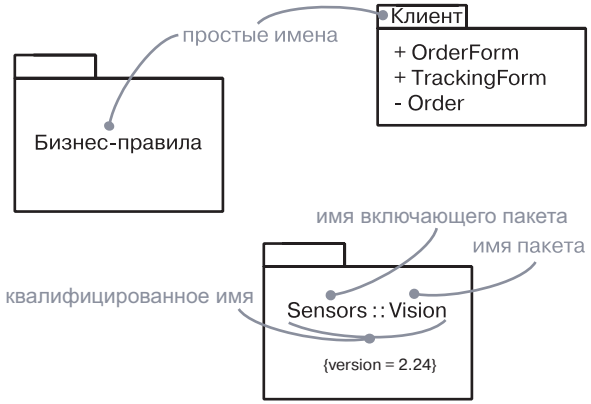


Рис. 12.2. Простые и квалифицированные имена пакетов

На заметку. Имя пакета может состоять из любых букв латинского алфавита и цифр, а также некоторых знаков препинания (за исключением таких, как двоеточие, которое применяется для разделения имен пакетов, один из которых входит в другой). Имя может занимать несколько строк (количество символов не ограничено). Обычно для именования пакета используют одно или несколько коротких существительных, взятых из словаря модели.

Элементы, принадлежащие пакету

Композиция обсуждается в главе 10.

Пакет может владеть другими элементами, в том числе классами, интерфейсами, компонентами, узлами, кооперациями, вариантами использования, диаграммами и даже прочими пакетами. *Владение* (ownership) – это связь композиции, означающая, что элемент объявлен внутри пакета. Если пакет удаляется, то уничтожается и принадлежащий ему элемент. Каждый элемент может принадлежать только одному пакету.

На заметку. Пакет владеет элементами модели, объявленными внутри него. Это могут быть, в частности, классы, ассоциации, обобщения, зависимости и примечания. Пакет не владеет элементами, на которые осуществляется лишь ссылка из него.

Пакет определяет свое пространство имен, то есть элементы одного вида должны иметь имена, уникальные в контексте включающего пакета. Например, в одном пакете не может быть двух классов `Queue` (Очередь), но может быть один класс `Queue` в пакете `P1`, а другой – в пакете `P2`. `P1::Queue` и `P2::Queue` имеют разные квалифицированные имена и поэтому являются различными классами.

На заметку. Во избежание недоразумений лучше не использовать одинаковые имена в различных пакетах.

Элементам различного вида можно присваивать одинаковые имена в пределах пакета. Так, допустимо наличие класса `Timer` (Таймер) и компонента `Timer` в одном и том пакете. Однако, чтобы предотвратить путаницу, лучше назначать всем элементам пакета уникальные имена.

Процесс импорта обсуждается ниже в данной главе.

Один пакет может содержать другие, а следовательно, допускается иерархическая декомпозиция модели. Например, может существовать класс `Camera` (Камера), принадлежащий пакету `Vision` (Оптическое Устройство), который, в свою очередь, содержится в пакете `Sensors` (Датчики). Полное имя этого класса – `Sensors::Vision::Camera`. Лучше, однако, избегать слишком глубокой вложенности пакетов: два-три уровня – предел управляемости. При необходимости для организации пакетов стоит использовать импорт, а не вложения.

Описанная семантика владения делает пакеты важным механизмом масштабирования системы. Без них пришлось бы создавать большие плоские модели, все элементы которой должны иметь уникальные имена. Такие конструкции были бы совершенно неуправляемы, особенно если входящие в модель классы и другие элементы созданы различными коллективами. Пакеты позволяют контролировать элементы, образующие систему, в процессе ее эволюции.

Как показывает рис. 12.3, содержание пакета можно представить графически или в текстовом виде. Обратите внимание, что если

вы изображаете принадлежащие пакету элементы, то имя пакета пишется внутри закладки. Впрочем, содержимое пакета не показывают таким образом, а применяют имеющиеся инструментальные средства, позволяющие раскрыть пакет и просмотреть его содержимое.

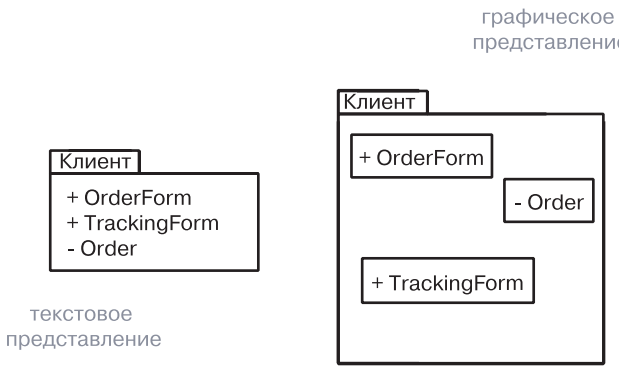


Рис. 12.3. Элементы, принадлежащие пакету

На заметку. Язык UML подразумевает наличие в любой модели анонимного корневого пакета, а значит, все элементы одного и того же вида на верхнем уровне модели должны иметь уникальные имена.

Видимость

Видимость описывается в главе 9.

Видимость принадлежащих пакету элементов можно контролировать так же, как видимость атрибутов и операций класса. По умолчанию такие элементы являются открытыми, то есть видимы для всех элементов, содержащихся в любом пакете, импортирующем данный. Защищенные элементы видимы только для потомков, а закрытые вообще невидимы вне своего пакета. Например, на рис. 12.3 `OrderForm` (БланкЗаказа) – это открытая часть пакета `Client` (Клиент), а `Order` (Заказ) – закрытая. Любой пакет, импортирующий данный, может «видеть» объект `OrderForm`, но «не видит» `Order`. При этом полное квалифицированное имя для `OrderForm` будет `Client::OrderForm`.

Видимость принадлежащего пакету элемента обозначается соответствующим символом перед его именем. Для открытых элементов используется знак + (плюс), как в случае с `OrderForm` на рис. 12.3. Все открытые части пакета в совокупности составляют его *интерфейс*.

Так же, как и в отношении классов, для имен защищенных элементов используют символ # (решетка), а для закрытых добавляют символ – (минус). Напомним, что защищенные элементы будут

видимы только для пакетов, наследующих данному, а закрытые вообще невидимы вне пакета, в котором объявлены.

Пакетная видимость показывает, что класс видим только для других классов, объявленных в том же самом пакете, но скрыт для классов, объявленных в других пакетах. Пакетная видимость изображается с помощью символа ~ (тильда) перед именем класса.

Импорт и экспорт

Предположим, в вашей модели есть два класса одного уровня, расположенных рядом друг с другом: `A` и `B`. Класс `A` «видит» `B` и наоборот, то есть любой из них может зависеть от другого. Оба они образуют тривиальную систему, и для них не надо создавать никаких пакетов.

Допустим теперь, что у вас имеется несколько сотен равноправных классов. Размер сети связей, которую вы можете «соткать» между ними, не поддается воображению. Более того, столь огромную группу неорганизованных классов просто невозможно воспринять в ее целостности. Это реальная проблема больших систем: простой неограниченный доступ не позволяет осуществить масштабирование. В таких случаях для организации абстракций приходится применять пакеты.

Связи зависимости обсуждаются в главе 5, механизмы расширения UML – в главе 6.

Итак, допустим, что класс `A` расположен в одном пакете, а класс `B` – в другом, причем оба пакета равноправны. Допустим также, что `A` и `B` объявлены открытыми частями в своих пакетах. Эта ситуация коренным образом отличается от двух предыдущих. Хотя оба класса объявлены открытыми, свободный доступ одного из них к другому невозможен, ибо границы пакетов непрозрачны. Однако если пакет, содержащий класс `A`, импортирует пакет-владелец класса `B`, то `A` сможет «видеть» `B`, хотя `B` по-прежнему не будет «видеть» `A`. Импорт дает элементам одного пакета односторонний доступ к элементам другого. На языке UML связь импорта моделируют как зависимость, дополненную стереотипом `import`. Упаковывая абстракции в семантически осмысленные блоки и контролируя доступ к ним с помощью импорта, вы можете управлять сложностью систем, насчитывающих множество абстракций.

На заметку. Фактически в рассмотренной ситуации применяются два стереотипа – `import` и `access`, и оба они показывают, что исходный пакет имеет доступ к элементам целевого. Стереотип `import` добавляет содержимое целевого пакета в открытое пространство имен исходного, поэтому нет необходимости в квалификации их имен. Таким образом, возникает вероятность конфликта имен, которого необходимо избегать, если вы

хотите, чтобы модель была хорошо согласована. Стереотип access добавляет содержимое целевого пакета в закрытое пространство имен исходного. Единственное различие заключается в том, что нельзя повторно экспортировать импортированные элементы, если какой-либо третий пакет импортирует первоначальный исходный пакет. Чаще используют стереотип import.

Интерфейсы обсуждаются в главе 11.

Открытые элементы пакета называют *экспортируемыми*. Так, на рис. 12.4 пакет GUI экспортирует два класса – Window (Окно) и Form (Форма). Класс EventHandler (ОбработчикСобытий) является защищенной частью пакета. Довольно часто экспортируемыми элементами пакета являются интерфейсы.

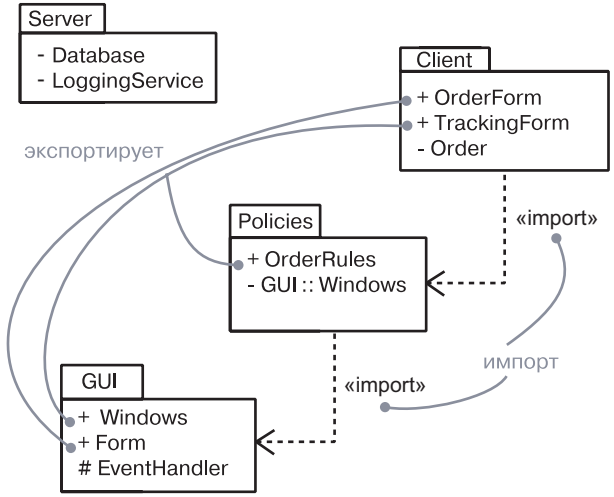


Рис. 12.4. Импорт и экспорт

Экспортируемые элементы будут видимы для содержимого тех пакетов, для которых видим данный пакет. В примере на рис. 12.4 пакет Policies (Политики) явно импортирует пакет GUI. Таким образом, классы GUI::Window и GUI::Form будут видимы для содержимого пакета Policies как классы с простыми именами Window и Form. Но класс GUI::EventHandler будет скрыт, так как является защищенным. С другой стороны, пакет Server не импортирует GUI, поэтому элементы Server имеют доступ к открытым элементам GUI, но для этого им понадобятся квалифицированные имена, например GUI::Window. Аналогично у GUI нет доступа к содержимому пакета Server даже с использованием квалифицированных имен, поскольку оно закрыто.

Зависимости импорта и доступа являются транзитивными. В данном примере пакет Client импортирует Policies, а Policies – GUI, так что можно сказать, что Client транзитивно импортирует GUI. Если Policies имеет доступ в GUI, не импортируя его, то Client не добавляет элементы GUI к своему пространству имен, но может обращаться к ним, используя квалифицированные имена (такие как, например, GUI::Window).

На заметку. Элемент, видимый внутри пакета, будет видим также и внутри всех вложенных в него пакетов. Вообще, вложенные пакеты могут «видеть» все, что «видят» включающие их пакеты. Имя во вложенном пакете может скрывать имя во включающем пакете; в этом случае для ссылки требуется квалифицированное имя.

Типичные приемы моделирования

Моделирование групп элементов

Чаще всего пакеты применяют для организации элементов моделирования в именованные группы, с которыми потом можно будет работать как с единым целым. Создавая простое приложение, можно вообще обойтись без пакетов, поскольку все ваши абстракции прекрасно разместятся в единственном пакете. В более сложных системах вы скорее обнаружите, что многие классы, компоненты, узлы, интерфейсы и даже диаграммы естественным образом разделяются на группы. Эти группы и моделируют в виде пакетов.

Между классами и пакетами есть одно значительное различие: классы являются абстракцией сущности из предметной области или из области решения, а пакеты – это механизмы организации таких сущностей в модели. Пакеты не видны в работающей системе, они служат исключительно механизмом организации разработки.

Пять представлений архитектуры обсуждаются в главе 2.

Чаще всего с помощью пакетов элементы одинаковых базовых типов организуют в группы. Например, классы и их связи в представлении системы с точки зрения проектирования можно разбить на несколько пакетов и контролировать доступ к ним с помощью зависимостей импорта. Компоненты представления реализации допустимо организовать таким же образом.

Пакеты также применяются для группирования элементов различных типов. Например, если система создается несколькими коллективами разработчиков, расположенными в разных местах, то пакеты можно использовать для управления конфигурацией, размещая в них все классы и диаграммы, так чтобы члены разных

коллективов могли независимо извлекать их из хранилища и помещать обратно. На практике пакеты часто применяют для группирования элементов модели и ассоциированных с ними диаграмм.

Чтобы смоделировать группы элементов, необходимо:

- Просмотреть элементы модели в некотором представлении архитектуры системы с целью поиска групп элементов, близких семантически или концептуально.
- Поместить каждую такую группу в пакет.
- Для каждого пакета определить, какие элементы должны быть доступны извне. Пометить их как открытые, а остальные – как защищенные или закрытые. В сомнительных случаях скрыть элемент.
- Явно соединить пакеты зависимостями импорта с теми пакетами, от которых они зависят.
- При наличии семейств пакетов соединить специализированные пакеты с более общими связями обобщения.

В качестве примера на рис. 12.5 показаны пакеты, которые организуют в классическую трехуровневую архитектуру классы, являющиеся частями представления информационной системы с точки зрения проектирования. Элементы пакета `User Services` (Пользовательские Сервисы) предоставляют визуальный интерфейс для ввода и вывода информации. Элементы пакета `Data Services` (Сервисы Данных) обеспечивают доступ к данным и их обновление. Пакет `Business Services` (Бизнес-сервисы) является связующим звеном между элементами первых двух пакетов; он охватывает все классы и другие элементы, отвечающие за выполнение бизнес-задачи, в том числе бизнес-правила, которые диктуют стратегию манипулирования данными.

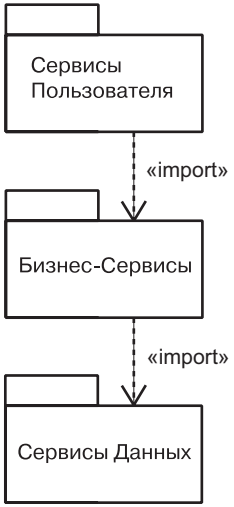


Рис. 12.5. Моделирование групп элементов

Документирующее помеченное значение обсуждается в главе 6.

Все абстракции простой системы можно поместить в один пакет. Однако, организуя классы и другие элементы представления системы с точки зрения проектирования в три пакета, вы не только сделаете модель более понятной, но и сможете контролировать доступ к ее элементам, скрывая одни и экспортируя другие.

На заметку. Изображая подобные модели, обычно показывают элементы, центральные для каждого пакета. Чтобы прояснить назначение пакетов, можно также использовать документирующее помеченное значение для каждого из них.

Моделирование архитектурных представлений

Пять представлений архитектуры описываются в главе 2.

Использование пакетов для группирования родственных элементов весьма важно – без него нельзя разработать сложную модель. Данный подход применим к организации таких элементов, как классы, интерфейсы, компоненты, узлы и диаграммы. Но при рассмотрении архитектурных представлений программных систем возникает потребность в еще более крупных блоках. Архитектурные представления тоже можно моделировать с помощью пакетов.

Связь представлений с моделями обсуждается в главе 32.

Напомним, что *представлением* (view) называется проекция организации и структуры системы, в которой внимание акцентируется на одном из конкретных ее аспектов. Из этого определения вытекают два следствия. Во-первых, систему можно разложить на почти ортогональные пакеты, каждый из которых имеет дело с набором архитектурно значимых решений (например, можно создать представления с точки зрения проектирования, взаимодействий, реализации, размещения и вариантов использования). Во-вторых, этим пакетам будут принадлежать все абстракции, относящиеся к данному представлению. Так, все компоненты модели принадлежат пакету, который моделирует представление реализации. В то же время пакеты могут содержать ссылки на элементы других пакетов.

Для моделирования архитектурных представлений необходимо:

- Идентифицировать набор архитектурных представлений, значимых в контексте вашей проблемы. На практике этот набор обычно включает представления проектирования, взаимодействия, реализации, размещения и вариантов использования.
- Разместить те элементы и диаграммы, которые необходимы и достаточны для визуализации, специфицирования, конструирования и документирования семантики каждого представления, в соответствующем пакете.

- При необходимости выполнить дальнейшую группировку этих элементов в пакеты.

Между элементами различных представлений обычно возникают зависимости. Таким образом, каждое представление на верхнем уровне системы должно быть открыто для всех остальных представлений этого уровня.

Моделирование систем обсуждается в главе 32.

В качестве примера на рис. 12.6 показана каноническая декомпозиция верхнего уровня, типичная для большинства сложных систем.

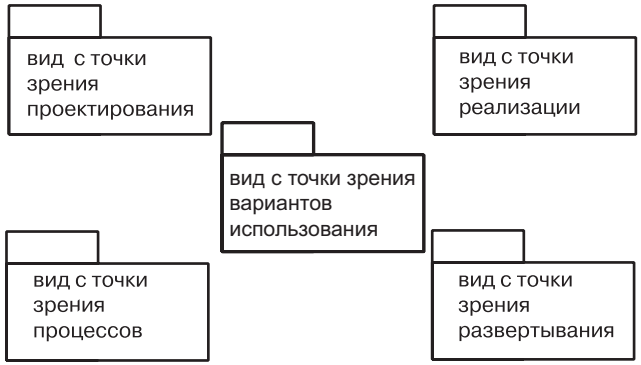


Рис. 12.6. Моделирование архитектурных представлений

Советы и подсказки

Моделируя пакеты в UML, помните, что они нужны только для организации элементов вашей модели. Если имеются абстракции, непосредственно материализуемые как объекты в системе, не пользуйтесь пакетами. Вместо них применяйте такие элементы моделирования, как классы или компоненты.

Хорошо структурированный пакет характеризуется следующими свойствами:

- он внутренне согласован и очерчивает четкую границу вокруг группы родственных элементов;
- он слабо связан и экспортирует в другие пакеты только те элементы, которые они действительно должны «видеть», а импортирует лишь те, которые принципиально важны для работы его собственных элементов;
- глубина вложенности пакета невелика, поскольку человек не способен воспринимать слишком глубоко вложенные структуры;
- владея сбалансированным набором элементов, пакет по отношению к другим пакетам в системе не должен быть ни слишком большим (если надо, расщепляйте его на более мелкие),

ни слишком маленьким (объединяйте элементы, которыми можно манипулировать как единым целым).

Изображая пакет в UML, руководствуйтесь следующими принципами:

- применяйте простую форму пиктограммы пакета, если не требуется явно раскрыть его содержимое;
- раскрывая содержимое пакета, показывайте только те элементы, которые принципиально необходимы для понимания его назначения в данном контексте;
- моделируя с помощью пакета сущности, относящиеся к управлению конфигурацией, раскрывайте значение меток, связанных с номерами версий.

Глава 13. Экземпляры

В этой главе:

- Экземпляры и объекты
- Моделирование конкретных экземпляров
- Моделирование прообраза экземпляров
- Реальный и абстрактный мир экземпляров

Описание внутренней структуры, которая предпочтительна в работе с прототипами объектов и ролями, приводится в главе 15.

Термины «экземпляр» и «объект» в большинстве случаев являются синонимами и часто бывают взаимозаменяемыми. Экземпляром называется конкретное воплощение абстракции, к которому может быть применен определенный набор операций и которое обладает состоянием для сохранения результатов этой операции.

Экземпляры используют для моделирования конкретных сущностей реального мира. Почти все строительные блоки UML вовлечены в дихотомию «класс/объект». Например, бывают варианты использования и экземпляры вариантов использования, узлы и экземпляры узлов, ассоциации и экземпляры ассоциаций и т.д.

Введение

Предположим, вы строите дом для своей семьи. Используя слово «дом», а не «автомобиль» или какое-нибудь другое понятие, вы заранее сужаете словарь, применяемый при решении задачи. Дом является абстракцией «постоянного или временного жилища, цель которого – предоставить убежище». Напротив, автомобиль – это «мобильное самодвижущееся устройство, предназначенное для перевозки людей с места на место». В ходе работы, состоящей в соединении различных, зачастую противоречивых требований, модель дома постепенно конкретизируется. Например, вы можете выбрать «дом с тремя спальнями и подвалом» – словом, уточнить детали.

Получив ключи от готового дома и войдя внутрь, вы погружаетесь в реальную обстановку – оцениваете здание не просто как дом с тремя спальнями, а как «мой дом с тремя спальнями, расположенный по такому то адресу». Если вы человек сентиментальный, то, возможно, дадите ему какое-нибудь имя, например «Святой приют».

Между «домом с тремя спальнями» и «моим домом с тремя спальнями, названным Святым приютом» существует фундаментальное различие. В то время как первый – лишь абстракция, описывающая определенный тип дома с различными свойствами, второй представляет собой конкретный экземпляр этой абстракции, воплощенный в материальной форме, и каждое его свойство имеет реальное значение.

Классы обсуждаются в главах 4 и 9, компоненты – в главе 15, узлы – в главе 27, варианты использования – в главе 17. Фактически вместо того, что мы в дальнейшем будем называть экземпляром, в UML используется термин «спецификация экземпляра», но это тонкость метамодели.

Дихотомия «класс/объект» обсуждается в главе 2.

Ассоциации обсуждаются в главах 5 и 10, ссылки – в главах 14 и 16.

Абстракция описывает идеальную суть предмета, экземпляр – его конкретное воплощение. Такое разделение на абстракцию и экземпляр обнаруживается во всем, что вы моделируете. У одной абстракции может быть сколько угодно экземпляров. Для данного экземпляра всегда существует абстракция, определяющая характеристики, общие для всех подобных экземпляров.

В языке UML можно представить как абстракции, так и их экземпляры. Почти все строительные блоки этого языка, в особенности классы, компоненты, узлы и варианты использования, могут быть промоделированы в терминах своей сущности или своих экземпляров. По большей части вы будете работать с ними как с абстракциями, но если захотите промоделировать конкретные воплощения, то придется иметь дело с экземплярами.

Графическое представление экземпляров показано на рис. 13.1. Эта нотация позволяет визуализировать как именованные, так и анонимные экземпляры.

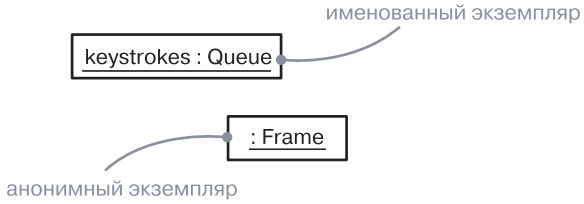


Рис. 13.1. Экземпляры

Базовые понятия

Экземпляром (instance) называется конкретное воплощение абстракции, к которому могут быть применены операции и которое обладает состоянием, сохраняющим их результаты. Понятия «экземпляр» и «объект» практически синонимичны. Экземпляр изображают с подчеркнутым именем.

На заметку. Обычно конкретное воплощение класса называют объектом. *Объекты* – это экземпляры классов; можно сказать, что все объекты являются экземплярами, но не все экземпляры – объектами. Например, экземпляр ассоциации не является объектом: это просто экземпляр, также называемый ссылкой. Впрочем, лишь самые дотошные создатели моделей обращают внимание на такие нюансы.

Абстракции и экземпляры

Классификаторы обсуждаются в главе 9.

Экземпляры не существуют сами по себе: они почти всегда связаны с абстракцией. На языке UML чаще всего моделируют экземпляры классов (называемые объектами), хотя также бывают экземпляры и других сущностей, таких как компоненты, узлы, варианты использования и ассоциации. В UML экземпляр легко отличить от абстракции – для этого надо просто подчеркнуть его имя.

Объект в общем случае занимает некоторое место в реальном или концептуальном мире, и над ним можно производить те или иные операции. Например, экземпляром узла обычно является компьютер, физически расположенный в некоем помещении; экземпляром компонента – файл, размещенный в том или ином каталоге; экземпляр записи о клиенте занимает какой-то объем оперативной памяти компьютера. Экземпляр траектории полета самолета тоже являет собой нечто конкретное, поддающееся обработке математическими методами.

Абстрактные классы обсуждаются в главе 9, интерфейсы – в главе 11.

С помощью UML допустимо моделировать не только непосредственные физические экземпляры, но и менее конкретные сущности. Например, абстрактный класс по определению не может иметь непосредственных экземпляров. Разрешается, однако, моделировать косвенные экземпляры абстрактных классов, чтобы показать, как данный класс можно использовать в качестве прототипа. Строго говоря, такого объекта не существует, но с практической точки зрения он позволяет поименовать любой потенциальный экземпляр конкретного потомка этого абстрактного класса. Тоже самое относится и к интерфейсам. Хотя они по определению не могут иметь непосредственных экземпляров, можно смоделировать экземпляр-прототип интерфейса, который будет представлять один из потенциальных экземпляров конкретных классов, реализующих данный интерфейс.

Диаграммы объектов обсуждаются в главе 14.

Моделируемые экземпляры помещают в диаграммы объектов, если надо показать их структурные детали, или в диаграммы взаимодействия и деятельности, если нужно визуализировать их участие в динамических ситуациях. Хотя обычно этого не требуется, их можно включать и в диаграммы классов, если надо показать связь объекта и его абстракции.

Типы экземпляров

У каждого экземпляра есть свой тип. *Тип экземпляра* должен быть его конкретным классификатором, но спецификация экземпляра (которая не является конкретным экземпляром) может иметь абстрактный тип. В текстовой строке сначала указывается тип, а затем через двоеточие – имя экземпляра (например, `t : Transaction`).

Диаграммы взаимодействия обсуждаются в главе 19, диаграммы деятельности – в главе 20; динамическая типизация рассматривается в главе 11, классификаторы – в главе 9.

Обычно классификатор экземпляра статичен. Например, после создания экземпляра класса последний не изменится на протяжении всего времени существования объекта. Однако, несмотря на это, в некоторых ситуациях моделирования и в некоторых компьютерных языках возможна смена абстракции экземпляра. К примеру, объект `Caterpillar` (Гусеница) может стать объектом `Butterfly` (Бабочка). По сути, это все тот же объект, но принадлежащий другой абстракции.

На заметку. На протяжении разработки возможно использование экземпляра без определенного классификатора, который должен быть изображен как объект, но с пропуском имени абстракции (см. пример на рис. 13.2). Использовать подобные объекты можно для моделирования весьма абстрактного поведения, хотя в конечном счете придется связать такие экземпляры с абстракцией, чтобы присвоить им необходимую семантику.

Имена

Операции обсуждаются в главах 4 и 9, компоненты – в главе 15, узлы – в главе 27.

Каждый экземпляр может обладать именем, отличающим его в данном контексте от остальных экземпляров. Обычно объект существует в контексте операции, компонента или узла. Имя представляет собой текстовую строку, например `t` или `myCustomer` (мойПокупатель) на рис. 13.2. Само по себе оно называется *простым* именем. Абстракция экземпляра может иметь как простое имя, скажем `Transaction` (Транзакция), так и *квалифицированное*, например `Multimedia::AudioStream` (Мультимедиа::Аудиопоток). Составное имя образуется путем добавления перед именем абстракции имени пакета, в котором она находится.

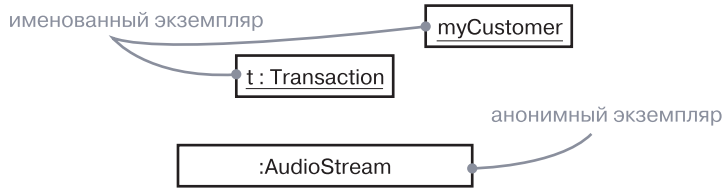


Рис. 13.2. Именованные и анонимные экземпляры

Объект можно именовать явно (`myCustomer` и т.п.), делая его имя осмысленным для пользователя. Можно дать ему простое имя (наподобие только что упомянутого) и скрыть его абстракцию, если она очевидна из контекста. Однако зачастую настоящее имя объекта

известно только компьютеру, который с ним работает. В таких случаях появляется анонимный объект (например, :AudioStream). Каждое появление анонимного объекта отлично от всех других его появлений. Если вы даже не знаете абстракцию объекта, то должны по крайней мере дать ему явное имя.

Роли и структурированные классы обсуждаются в главе 15.

Имя и тип объекта записываются в одну строку, например t : Transaction. В случае с объектом, в отличие от роли в структурированном классе, вся строка подчеркнута.

На заметку. Имя экземпляра может состоять из любых букв латинского алфавита, цифр и некоторых знаков препинания (за исключением таких, как двоеточие, которое применяется для отделения имени экземпляра от имени его абстракции). Имя может занимать несколько строк (количество символов не ограничено). На практике для именования экземпляров используют одно или несколько коротких существительных, взятых из словаря моделируемой системы. При этом каждое слово, кроме первого, обычно пишется с прописной буквы (например, t или myCustomer).

Операции

Операции описаны в главах 4 и 9, полиморфизм – в главе 9.

Объект не просто занимает определенное место в реальном мире – он поддается неким манипуляциям. Операции, выполняемые объектом, объявляются в его абстракции. Например, если класс Transaction (Транзакция) содержит операцию commit (совершить) и у него имеется экземпляр t : Transaction, то можно написать выражение t.commit(). Его выполнение означает, что над объектом t осуществляется операция commit. В зависимости от связанной с этим классом иерархии наследования данная операция может быть вызвана полиморфно.

Состояние

Кроме прочего, объект обладает *состоянием*, под которым подразумевается совокупность всех его свойств и их текущих значений (включая также ссылки и связанные объекты, в зависимости от точки зрения). В число свойств входят атрибуты и ассоциации объекта, а также все его агрегированные части. Таким образом, состояние объекта динамично, и при его визуализации вы фактически описываете значение его состояния в данный момент времени и в данной точке пространства. Процесс изменения состояния объекта можно изобразить графически, если на одной и той же диаграмме

взаимодействия нарисовать его несколько раз, причем на каждом рисунке будет отражено новое состояние.

Атрибуты обсуждаются в главе 4, диаграммы взаимодействия – в главе 19. Альтернативный способ описать процесс изменения состояния объекта – использование автоматов – рассматривается в главе 22.

Совершая над объектом операцию, вы изменяете его состояние, однако при запросе объекта его состояние не меняется. Предположим, бронируя билет на самолет (объект r : Заказ), вы определяете значение одного из атрибутов (например, цена билета = 395.75). Изменив впоследствии условия заказа – скажем, добавив к маршруту еще одну пересадку, – вы тем самым изменяете и его состояние: например, цена билета становится равной 1024.86.

На рис. 13.3 показано, как изображать значение атрибутов объекта средствами языка UML. Значение атрибута id объекта myCustomer равно "432-89-1783". В данном случае тип идентификатора (SSN – номер социального страхования) показан явно, хотя его можно и опустить (как это сделано для атрибута active = True), поскольку тип содержится в объявлении id в ассоциированном классе объекта myCustomer.

С классом можно ассоциировать также автомат, что особенно полезно при моделировании управляемых событиями систем или жизненного цикла класса. В таких случаях можно показать состояние автомата для данного объекта в данный момент времени. Состояние изображается в квадратных скобках после типа. Например, на рис. 13.3 показано, что объект c – экземпляр класса Phone (Телефон) находится в состоянии WaitingForAnswer (ЖдетОтвета), определенном в автомате для класса Phone.

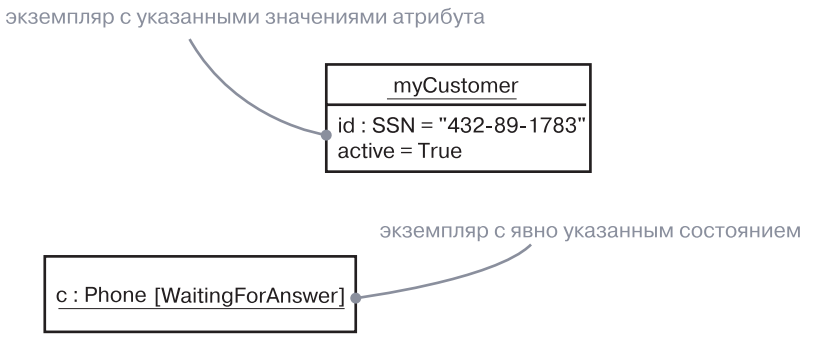


Рис. 13.3. Состояние объекта

На заметку. Так как объект может одновременно находиться в нескольких состояниях, вы можете вывести на диаграмме весь список его текущих состояний.

Другие свойства

Процессы и потоки обсуждаются в главе 23.

Поскольку процессы и потоки являются важными составными частями представления системы с точки зрения процессов, в UML имеется графический образ для различения активных и пассивных элементов (элемент считается активным, если он является частью процесса или потока и представляет собой исходную точку потока управления). Вы можете объявить активные классы, материализующие процесс или поток, и, соответственно, выделить экземпляр активного класса (см. рис. 13.4).

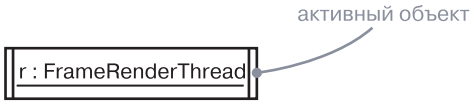


Рис. 13.4. Активные объекты

Диаграммы взаимодействия обсуждаются в главе 19.

На заметку. Чаще всего активные объекты применяются в контексте диаграмм взаимодействия, моделирующих множественные потоки управления. Каждый активный объект представляет исходную точку потока управления и может использоваться для именования различных потоков.

Ссылки описаны в главах 14 и 16, статические атрибуты и операции – в главе 9.

В UML существуют еще два элемента, у которых могут быть экземпляры. Первый – это ассоциация. Экземпляром ассоциации является *ссылка* (link). Ссылка представляет собой семантическое соединение между объектами. Как и ассоциация, она изображается в виде линии; от ассоциации ее можно отличить по тому, что она соединяет только объекты. Второй вариант – статический атрибут, область действия которого ограничена классом. Статический атрибут – это, по сути дела, объект в этом классе, совместно используемый всеми экземплярами класса. В объявлении класса такой атрибут подчеркивается.

Стандартные элементы

Механизмы расширения UML обсуждаются в главе 6.

Все механизмы расширения языка UML применяются к объектам. Тем не менее экземплярам обычно не приписывают стереотип непосредственно и не связывают с ними помеченных значений. Вместо этого стереотипы и помеченные значения объекта выводятся из ассоциированных с ним абстракций. В качестве примера на рис. 13.5 показано, как можно явно приписать стереотип самому объекту или его абстракции.

Рис. 13.5. Объекты со стереотипами

В UML определены два стандартных стереотипа, применимых к связям зависимости между объектами и классами:

- `instanceOf` – показывает, что объект-клиент является экземпляром классификатора-сервера. Это редко изображается графически. Обычно используется текстовая запись с двоеточием;
- `instantiate` – показывает, что класс-клиент создает экземпляры классификатора-сервера.

Типичные приемы моделирования

Моделирование конкретных экземпляров

Моделируя конкретные экземпляры, вы визуализируете сущности, имеющиеся в реальном мире. Конечно, вы не увидите экземпляр класса `Client`, если данный клиент не стоит перед вами, но, по крайней мере, имеете возможность увидеть в отладчике представление этого объекта.

Диаграммы компонентов обсуждаются в главе 15, диаграммы размещения – в главе 31, диаграммы объектов – в главе 14.

Одна из сфер применения объектов – моделирование конкретных экземпляров, существующих в реальном мире. Например, моделируя топологию сети вашей организации, вы пользуетесь диаграммами размещения, содержащими экземпляры узлов. Аналогичным образом, если вы хотите моделировать компоненты, расположенные в физических узлах сети, то будете пользоваться диаграммами компонентов, которые содержат их экземпляры. Наконец, если к вашей системе подключен отладчик, вы сможете представить структурные связи между экземплярами с помощью диаграмм объектов.

При моделировании конкретных экземпляров необходимо соблюдать такую последовательность действий:

- ❑ идентифицировать экземпляры, необходимые и достаточные для визуализации, специфицирования, конструирования или документирования моделируемой задачи;
- ❑ изобразить эти объекты как экземпляры с помощью UML. Если можно, дать каждому из них собственное имя. Если для объекта не существует осмысленного имени, изобразить его как анонимный объект;

- выявить для каждого экземпляра стереотипы, помеченные значения и атрибуты (вместе с их значениями), необходимые и достаточные для моделирования задачи;
- изобразить эти экземпляры и связи между ними на диаграмме объектов или на другой диаграмме, соответствующей типу экземпляра.

В качестве примера на рис. 13.6 показана диаграмма объектов, взятая из системы проверки подлинности кредитных карточек. Такой ее можно увидеть в отладчике, тестирующем приложения.

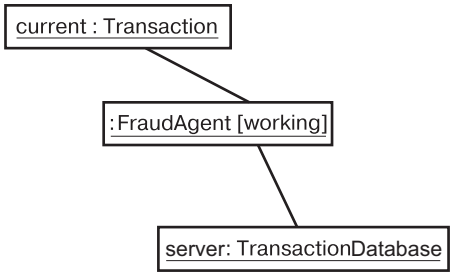


Рис. 13.6. Моделирование конкретных экземпляров

Советы и подсказки

Моделируя экземпляры на языке UML, помните, что каждый из них должен означать конкретное воплощение некоторой абстракции (обычно – класса, компонента, узла, варианта использования или ассоциации). Хорошо структурированный экземпляр обладает следующими свойствами:

- явно ассоциирован с конкретной абстракцией;
- имеет уникальное имя, взятое из словаря предметной области или области решения.

Изображая экземпляры в UML, руководствуйтесь следующими принципами:

- всегда показывайте имя абстракции, которой принадлежит экземпляр, если это не очевидно из контекста;
- показывайте стереотип, роль или состояние экземпляра, только если это необходимо для понимания объекта в данном контексте;
- организуйте длинные списки атрибутов экземпляра, группируя их вместе с их значениями в соответствии с категориями.

Глава 14. Диаграммы объектов

В этой главе:

Диаграммы объектов позволяют моделировать экземпляры сущностей, которые содержатся в диаграммах классов. На диаграмме объектов показано множество объектов и связей между ними в некоторый момент времени.

Диаграммы объектов применяют при моделировании статических представлений системы с точки зрения проектирования и процессов. При этом моделируется «снимок» системы в данный момент времени и изображается множество объектов, их состояний и связей между ними.

Диаграммы объектов важны не только для визуализации, специфицирования и документирования структурных моделей, но и для конструирования статических аспектов системы с помощью прямого и обратного проектирования.

Введение

Для человека, не знакомого с правилами игры, футбол может показаться чрезвычайно простым видом спорта: толпа народа беспорядочно носится по полю, гоняя мяч. Неискушенный зритель вряд ли увидит в этом движении какой-либо порядок или оценит красоту игры.

Если прервать матч и показать зрителю, какие роли исполняют отдельные игроки, перед ним предстанет совсем другая картина. В общей массе он различит нападающих, защитников и полузащитников, а понаблюдав за ними, поймет, как они взаимодействуют согласно определенной стратегии, направленной на то чтобы забить гол в ворота противника: ведут мяч по полю, отбирают его друг у друга и атакуют. В опытной команде вы никогда не найдете игроков, беспорядочно и бесцельно перемещающихся по полю. Напротив, в любой момент времени расположение игроков и их взаимодействия точно рассчитаны.

То же самое касается визуализации, специфицирования, конструирования и документирования программных систем. Стараясь проследить за потоком управления в работающей системе, вы быстро потеряете общее представление о том, как организованы ее составляющие части, особенно если имеется несколько потоков. Точно так же изучение состояния одного объекта в конкретный момент времени не поможет понять сложную структуру данных. Чтобы решить эту проблему, придется рассмотреть не только сам объект, но и его ближайших соседей и связи между ними. Вообще, во всех объектно-ориентированных системах, за исключением самых простых, объекты не существуют автономно, а вполне определенным образом связаны со множеством других объектов. Более того, неполадки в таких системах чаще всего объясняются не логическими ошибками, а именно нарушениями взаимосвязей объектов или непредвиденными изменениями их состояния.

Диаграммы классов обсуждаются в главе 8, взаимодействия – в главе 16, диаграммы взаимодействия – в главе 19.

В языке UML статические аспекты строительных блоков системы визуализируют с помощью диаграмм классов. Диаграммы взаимодействия позволяют увидеть динамические аспекты системы, включая экземпляры этих строительных блоков и сообщения, которыми они обмениваются. Диаграмма объектов содержит множество экземпляров сущностей, представленных на диаграмме классов. Таким образом, диаграммы объектов представляют статическую составляющую взаимодействия и состоят из взаимодействующих объектов, однако сообщения на них не показаны. Диаграмма объектов отражает состояние системы в фиксированный момент времени, как показано на рис. 14.1.

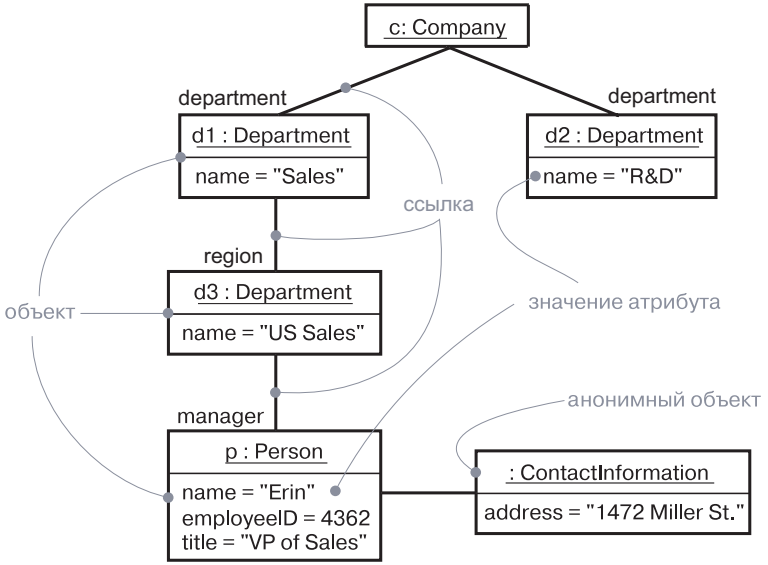


Рис. 14.1. Диаграмма объектов

Базовые понятия

На *диаграмме объектов* (object diagram) показаны объекты и их связи в некоторый момент времени. Ее представляют в виде графа, состоящего из вершин и ребер.

Общие свойства

Общие свойства диаграмм обсуждаются в главе 7.

У диаграммы объектов, как и у любой диаграммы, есть имя и графическое содержание, являющееся проекцией модели. От прочих диаграмм отличается своим конкретным наполнением.

Содержание

Объекты обсуждаются в главе 13, ссылки – в главе 16.

Диаграммы объектов, как правило, содержат объекты и ссылки, а кроме того, подобно всем прочим диаграммам, могут включать в себя примечания и ограничения. Иногда в них помещают и классы, особенно если надо визуализировать классы, стоящие за каждым экземпляром.

Диаграммы классов обсуждаются в главе 8, диаграммы взаимодействия – в главе 19.

На заметку. Диаграмма объектов соотносится с диаграммой классов: последняя содержит описание общей ситуации, а первая – описание конкретных экземпляров, выводимых из диаграммы классов. На диаграмме объектов представлены прежде всего объекты и ссылки. Диаграммы размещения также встречаются в двух формах: общей форме и форме экземпляров. В первом случае они описывают типы узлов, а во втором – конкретную конфигурацию экземпляров узлов, описываемых этими типами.

Типичное применение

Представления проектирования обсуждаются в главе 2.

С помощью диаграмм объектов, как и с помощью диаграмм классов, моделируют статическое представление системы с точки зрения проектирования или процессов, но принимая во внимание реальные экземпляры или прототипы. Это представление отражает главным образом функциональные требования к системе, то есть услуги, которые она должна предоставлять конечным пользователям. Диаграммы объектов позволяют моделировать статические структуры данных.

При моделировании статического представления системы с точки зрения проектирования или взаимодействия диаграммы объектов обычно применяют для моделирования структуры объектов.

Диаграммы взаимодействия обсуждаются в главе 19.

Моделирование структуры объектов предполагает получение «снимка» объектов системы в данный момент времени. Диаграмма объектов представляет один статический кадр в динамическом сценарии, описываемом диаграммой взаимодействия. Они применяются для визуализации, специфицирования, конструирования и документирования определенных экземпляров в системе, а также связей между этими экземплярами. Динамику поведения можно изобразить в виде последовательности кадров.

Типичные приемы моделирования

Моделирование структур объектов

Конструируя диаграмму классов, компонентов или размещения, вы на самом деле описываете группу интересующих вас абстракций и раскрываете в данном контексте их семантику и связи с другими абстракциями в группе. Эти диаграммы отражают только потенциальные возможности. Например, если класс `A` связан с классом `B` ассоциацией типа «один-ко-многим», то с одним экземпляром класса `A` может быть связано пять экземпляров класса `B`, а с другим – только один. Кроме того, в любой конкретный момент времени экземпляр класса `A` и связанные с ним экземпляры класса `B` будут иметь вполне определенные значения своих атрибутов и состояния автоматов.

«Заморозив» работающую систему или просто представив себе некий миг в жизни моделируемой системы, вы обнаружите совокупность объектов, каждый из которых находится в определенном состоянии и имеет конкретные связи с другими объектами. Диаграммы объектов позволяют визуализировать, специфицировать, конструировать и документировать структуру, образуемую этими объектами. Особенно полезны они бывают при моделировании сложных структур данных.

При моделировании вида системы с точки зрения проектирования при помощи набора диаграмм классов можно полностью определить семантику абстракций и их связей. Однако диаграммы объектов не позволяют полностью описать объектную структуру системы. У класса может быть большое количество различных экземпляров, а при наличии нескольких классов, связанных друг с другом, число возможных конфигураций объектов многократно возрастает. Поэтому при использовании диаграмм объектов нужно сосредоточиться на изображении интересующих вас наборов конкретных объектов или объектов-прототипов. Именно это и понимается под *моделированием структуры объектов* – отображение на диаграмме множества объектов и отношений между ними в некоторый момент времени.

Для моделирования структуры объектов понадобится:

- Идентифицировать механизм, который вы собираетесь моделировать. *Механизм* представляет собой некоторую функцию или поведение части моделируемой системы, являющееся результатом взаимодействия сообщества классов, интерфейсов и других сущностей.
- Для каждого обнаруженного механизма идентифицировать классы, интерфейсы и другие элементы, участвующие в кооперации, а также связи между ними.
- Рассмотреть один из сценариев работы механизма. «Заморозить» этот сценарий в некоторый момент времени и изобразить все объекты, участвующие в механизме.
- Показать состояние и значения атрибутов каждого такого объекта, если это необходимо для понимания сценария.
- Также показать ссылки между этими объектами, которые представляют экземпляры ассоциаций.

В качестве примера на рис. 14.2 показана совокупность объектов, взятая из реализации автономного робота. Внимание здесь акцентировано на нескольких объектах, составляющих часть механизма робота, предназначенного для расчета модели окружающей среды, в которой тот перемещается. Разумеется, в работе системы принимает участие гораздо больше объектов, но на данной диаграмме рассматриваются только абстракции, непосредственно вовлеченные в процесс формирования представления окружающей среды.

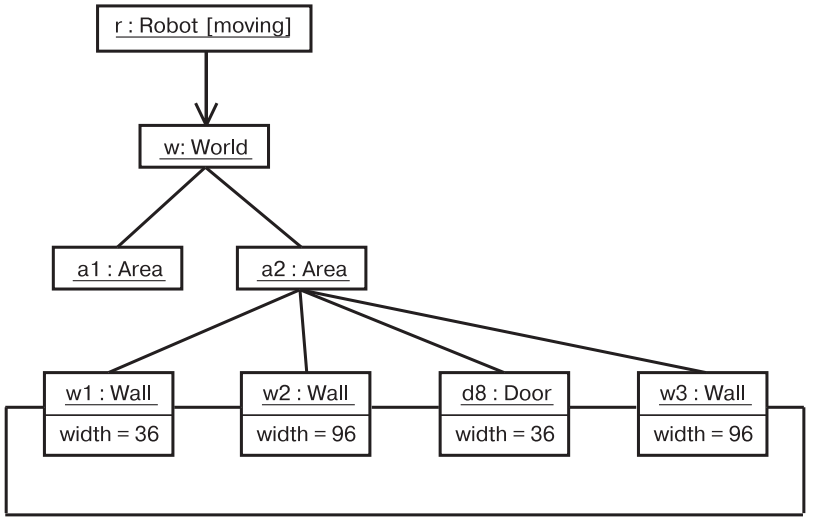


Рис. 14.2. Моделирование структур объектов

Как видно из рисунка, один из объектов соответствует самому роботу (r, экземпляр класса Robot) и в настоящий момент находится в состоянии moving (движется). Этот объект связан с экземпляром w класса World (Мир), являющегося абстракцией модели мира робота. В свою очередь объект w связан с мультиобъектом, который состоит из экземпляров класса Element (Элемент), описывающего сущности, опознанные роботом, но еще не включенные в его модель мира. Такие элементы помечены как части глобального состояния робота.

В текущий момент времени экземпляр w связан с двумя экземплярами класса Area. У одного из них (a2) показаны его собственные ссылки на объекты класса Wall (Стена) и объект класса Door (Дверь). Указана ширина каждой из трех стен и отмечено, что каждая связана с соседними. Как видно из диаграммы, робот распознал, что замкнутое помещение, в котором он находится, имеет с трех сторон стены, а с четвертой – дверь.

Обратное проектирование

Вам может пригодиться *обратное проектирование диаграммы объектов*, то есть создание модели на основе кода. Фактически, отлаживая вашу систему, вы непосредственно или с помощью каких-либо инструментов непрерывно осуществляете этот процесс. Например, чтобы отыскать «висячую» связь, необходимо реально или мысленно нарисовать диаграмму взаимодействующих объектов, которая и позволит определить, в каком месте оказалось некорректным состояние одного из них или нарушились его связи с другими.

Обратное проектирование диаграммы объектов необходимо вести по такой схеме:

- Выбрать, что именно вы хотите реконструировать. Обычно контекстом является какая-либо операция или экземпляр конкретного класса.
- С помощью инструментальных средств или просто пройдясь по сценарию, зафиксировать работу системы в некоторый момент времени.
- Идентифицировать множество интересующих вас объектов, взаимодействующих в данном контексте, и изобразить их на диаграмме объектов.
- Если это необходимо для понимания семантики, показать состояния объектов.
- Чтобы обеспечить понимание семантики, идентифицировать ссылки, существующие между объектами.

- Если диаграмма оказалась слишком сложной, упростить ее, убрав объекты, несущественные для прояснения данного сценария. Если диаграмма слишком проста, включить в нее окружение некоторых представляющих интерес объектов и подробней показать состояние каждого объекта.

Советы и подсказки

Создавая диаграммы объектов на языке UML, помните, что каждая такая диаграмма – это всего лишь графическое изображение статического представления системы с точки зрения проектирования или процессов. Ни одна отдельно взятая диаграмма объектов не в состоянии передать всю заключенную в этих представлениях информацию. На самом деле во всех системах, кроме самых тривиальных, существуют сотни, а то и тысячи объектов, большая часть которых анонимна. Полностью специфицировать все объекты системы и все способы, которыми они могут быть ассоциированы, невозможно. Следовательно, диаграммы объектов должны отражать только некоторые конкретные объекты или прототипы, входящие в состав работающей системы.

Хорошо структурированная диаграмма объектов характеризуется следующими свойствами:

- акцентирует внимание на одном аспекте статического представления системы с точки зрения проектирования или процессов;
- представляет лишь один из кадров динамического сценария, показанного на диаграмме взаимодействия;
- содержит только существенные для понимания данного аспекта элементы;
- уровень ее детализации соответствует уровню абстракции системы (показывайте только те значения атрибутов и дополнения, которые существенны для понимания);
- не настолько лаконична, чтобы пользователь упустил из виду нечто важное.

Создавая диаграмму объектов, придерживайтесь следующих правил:

- присваивайте ей имя, соответствующее назначению;
- располагайте ее элементы так, чтобы минимизировать пересечение линий;
- организуйте ее элементы так, чтобы семантически близкие сущности оказывались рядом;
- используйте примечания и цвет для привлечения внимания к важным особенностям диаграммы;
- включайте в описания каждого объекта значения и состояния, если это необходимо для понимания ваших намерений.

Глава 15. Компоненты

В этой главе:

- Компоненты, интерфейсы и реализация
- Внутренние структуры, порты, части и коннекторы
- Соединение подкомпонентов
- Моделирование API

Компонент – это логическая замещаемая часть системы, которая соответствует некоторому набору интерфейсов и обеспечивает их реализацию.

Хорошие компоненты определяют четкие абстракции с хорошо определенными интерфейсами, что дает возможность легко заменять старые компоненты на совместимые с ними новые.

Интерфейсы соединяют логические модели с моделями дизайна. Так, например, вы можете специфицировать интерфейс класса в логической модели, и тот же интерфейс будет поддерживаться некоторым компонентом дизайна, реализующим его.

Интерфейсы позволяют вам реализовывать компонент с использованием более мелких компонентов путем соединения их портов.

Введение

Раз уж вы планируете построить дом, то наверняка вам удастся выделить немного сбережений на домашний кинотеатр. Можно купить единственный блок, который включает телевизионный экран, тюнер, VCR/DVD-плейер и аудиоколонки. Такую систему легко установить, и работать она будет отлично, если отвечает вашим потребностям. Однако система, состоящая из единого блока, не слишком гибка. Вы вынуждены довольствоваться различным оборудованием именно в том сочетании, которое предлагает производитель. Возможно, вам не нужны высококачественные колонки и вы хотите установить новый телеэкран с высоким разрешением – тогда придется отказаться от всего блока, включая плейер, который, возможно, вполне вас устраивает.

Вы можете вставить штекер усилителя в выход, предназначенный для видеоаппаратуры, потому что и в том, и в другом случае используются одинаковые разъемы. Преимущество программных систем в том, что у них может быть неограниченное число «разъемов».

Более гибкий подход к установке домашнего кинотеатра – использование отдельных компонентов с разной функциональностью. На мониторе вы видите изображение; колонки воспроизводят звук, причем разместить их можно в любой точке комнаты, добиваясь хорошей акустики; тюнер и VCR/DVD-плейер тоже представлены в виде отдельных узлов, подобранных согласно вашим потребностям и финансовым ресурсам. Не будучи жестко связанными друг с другом, эти компоненты позволяют вам разместить их по своему усмотрению и соединить кабелями. Каждый кабель имеет особый тип разъема, подходящий к соответствующему порту устройства, поэтому вы не сможете, например, подключить колонку к видеовыходу. Зато при желании вы легко добавите в эту систему ваш старый проигрыватель. Если же возникнет идея ее обновить, вы будете заменять по одному компоненту, не поступаясь всеми прочими. Итак, с одной стороны, обеспечена большая гибкость, а с другой – высокое качество, которое можно постепенно «наращивать», если вы можете это себе позволить.

То же самое касается программного обеспечения. Когда вы конструируете приложение в виде большого монолитного узла, он получается «жестким», то есть впоследствии его трудно модифицировать. К тому же вы лишены возможности свободно распоряжаться имеющимися средствами. Если даже существующая система обладает большей частью необходимой вам функциональности, она, скорее всего, включает в себя кучу деталей, которые вам совершенно не нужны, однако их сложно или вообще невозможно удалить. Решение этой проблемы мы уже обрисовали выше на примере домашнего кинотеатра: следует строить системы из отдельных компонентов, которые гибко взаимодействуют друг с другом и при изменении требований могут удаляться и добавляться без ущерба для целого.

Базовые понятия

Интерфейсы обсуждаются в главе 11, классы – в главах 4 и 9.

Интерфейс – это набор операций, которые специфицируют сервис, предоставляемый либо требуемый классом или компонентом.

Компонент – замещаемая часть системы, которая соответствует набору интерфейсов и обеспечивает его реализацию.

Порт – специфическое «окно» в инкапсулированный компонент, принимающее сообщения для компонента и от него в соответствии с заданным интерфейсом.

Внутренняя структура – реализация компонента, представленная набором частей, соединенных друг с другом конкретным способом.

Часть – спецификация роли, составляющей часть реализации компонента. В экземпляре компонента присутствует экземпляр, соответствующий части.

Коннектор – связь коммуникации между двумя частями или портами в контексте компонента.

Компоненты и интерфейсы

Интерфейсы обсуждаются в главе 11.

Итак, интерфейс – набор операций, используемый для спецификации сервиса класса или компонента. Связь между компонентом и интерфейсом имеет важное значение. Все основанные на компонентах средства операционных систем (такие, как COM+, CORBA и Enterprise Java Beans) используют интерфейсы в качестве элементов, связывающих компоненты друг с другом.

Моделирование распределенных систем обсуждается в главе 24.

Чтобы сконструировать систему на основе компонентов, необходимо произвести ее декомпозицию, специфицируя интерфейсы, которые представляют основные соединения. Затем понадобится определить компоненты, реализующие интерфейсы, вместе с другими компонентами, которые имеют доступ к сервисам первых посредством своих интерфейсов. Этот механизм позволяет вам развернуть систему, сервисы которой в определенной мере независимы от местоположения и, как будет показано в следующем разделе, заменяемы.

Реализация обсуждается в главе 10.

Интерфейс, который реализован компонентом, называется *предоставляемым* (то есть данный компонент предоставляет интерфейс в виде сервиса другим компонентам). Компонент может декларировать множество предоставляемых интерфейсов. Интерфейс, который он использует, называется *требуемым*: ему соответствует данный компонент, когда запрашивает сервисы от других компонентов. Компонент может соответствовать множеству требуемых интерфейсов. Бывают компоненты, которые одновременно предоставляют и требуют интерфейсы.

Как показано на рис. 15.1, компонент изображается в виде прямоугольника с двузубчатой пиктограммой в правом верхнем углу. Внутри прямоугольника указывается имя компонента. У него могут быть атрибуты и операции, которые, впрочем, на диаграммах часто опускаются. Компонент может показывать сеть внутренней структуры, о чем пойдет речь чуть ниже.

Связь между компонентом и его интерфейсами выражается одним из двух способов. Первый более употребителен – интерфейс изображается в сокращенной, пиктографической форме. Предоставляемый интерфейс выглядит как кружок, соединенный линией с компонентом («леденец на палочке»). Требуемый интерфейс – полукруг, так же соединенный с компонентом («гнездо»). В обоих

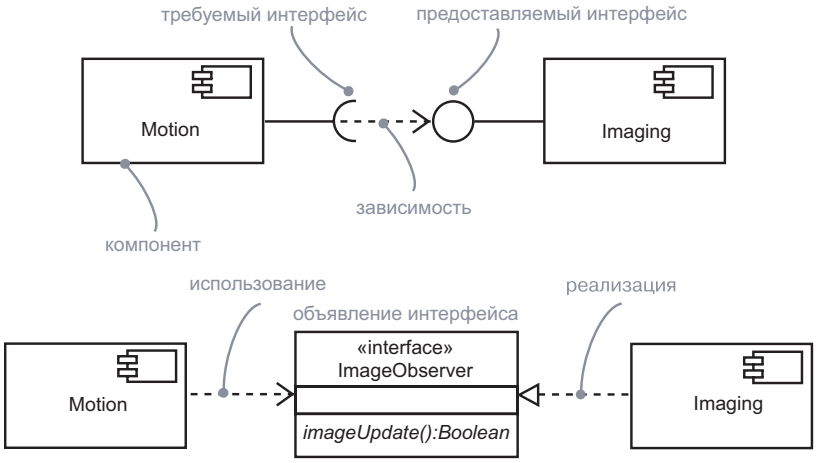


Рис. 15.1. Компоненты и интерфейсы

случаях имя интерфейса указано рядом с символом. Второй способ – изображение интерфейса в его расширенной форме (возможно, с указанием его операций). Компонент, который реализует интерфейс, соединяется с ним посредством полной связи реализации.

Компонент, который имеет доступ к сервисам другого компонента через интерфейс, соединяется с интерфейсом связью зависимости.

Определенный интерфейс может быть предоставлен одним компонентом и затребован другим. Поскольку этот интерфейс находится между двумя компонентами, их прямая зависимость друг от друга разрушается. Компонент, использующий данный интерфейс, будет работать правильно независимо от того, какой компонент его реализует. Конечно, компонент может быть использован в этом контексте тогда и только тогда, когда все его требуемые интерфейсы реализованы в качестве предоставляемых интерфейсов в других компонентах.

На заметку. Интерфейсы применяются на многих уровнях, как и другие элементы. Интерфейс уровня дизайна, который используется или реализован компонентом, на уровне реализации будет отображен на интерфейс, используемый или реализованный артефактом, который воплощает этот компонент.

Заменяемость

Артефакты обсуждаются в главе 26.

Основное назначение каждого средства операционной системы, основанной на компонентах, – обеспечить сборку системы из бинарных заменяемых артефактов. Это значит, что вы можете проектировать систему, используя компоненты и затем реализуя их в виде артефактов. Вы можете даже развивать систему, добавляя новые

компоненты и заменяя старые, но не перестраивая всю ее целиком. Интерфейсы – ключевое средство, обеспечивающее такие возможности. В работающей системе допускается применение любых артефактов, которые реализуют компоненты, согласованные и предоставляющие нужный интерфейс. Расширение системы возможно за счет создания компонентов, предоставляющих новые сервисы через другие интерфейсы, которые прочие компоненты, в свою очередь, могут обнаружить и использовать. Эта семантика проясняет цели определения компонентов в UML. Компонент соответствует набору интерфейсов и обеспечивает его реализацию, что позволяет замещать его – как в логическом дизайне, так и в основанной на нем физической реализации.

Компонент *замещаем* – это значит, что его можно заменять другим компонентом, который соответствует тем же интерфейсам (в процессе проектирования вы выбираете иной компонент вместо данного). Обычно механизм вставки и замены артефакта в исполнимой системе прозрачен для пользователя компонента и допускается объектными моделями (такими как COM+ и Enterprise Java Beans), которые требуют небольшой промежуточной трансформации, или осуществляется инструментами, автоматизирующими этот механизм.

Системы и подсистемы обсуждаются в главе 32.

Компонент – это часть системы – он редко используется сам по себе. Чаще он объединен с другими компонентами, то есть вовлечен в архитектурный или технологический контекст, где предполагается его использовать. Компонент логически и физически согласован и, таким образом, представляет структурный и/или поведенческий фрагмент более крупной системы. Во множестве систем он может быть использован повторно. Таким образом, компонент представляет собой фундаментальный строительный блок, на основе которого может быть спроектирована и составлена система. Это определение рекурсивно: то, что на одном уровне абстракции является системой, может быть компонентом на другом, более высоком уровне.

Наконец, как уже отмечалось в предыдущих разделах, компонент соответствует набору интерфейсов и обеспечивает его реализацию.

Организация компонентов

Пакеты обсуждаются в главе 12. Связи обсуждаются в главах 5 и 10.

Вы можете организовать компоненты тем же способом, что и классы, – группируя их в пакеты.

Кроме того, допускается организация компонентов путем установления между ними связей зависимости, обобщения, ассоциации (включая агрегацию) и реализации.

Одни компоненты могут быть построены из других. Об этом пойдет речь чуть ниже, в разделе «Внутренняя структура».

Порты

Интерфейсы удобны для описания общего поведения компонента, но им не присуща «индивидуальность»: реализация компонента должна лишь гарантировать, что все операции во всех предоставляемых интерфейсах реализованы. Для более полного контроля над реализацией можно использовать порты.

Порт (port) – это своеобразное «окно» в инкапсулированный компонент. Все взаимодействие с таким компонентом на входе и на выходе происходит через порты. Выражаемое внешне поведение компонента представляет собой сумму его портов – ни более ни менее. Вдобавок к этому порт наделен уникальностью. Один компонент может взаимодействовать с другим через определенный порт. При этом их коммуникации полностью описываются интерфейсами, которые поддерживает порт, даже если компонент поддерживает другие интерфейсы. В реализации внутренние части компонента могут взаимодействовать друг с другом через специфический внешний порт, поэтому каждая часть может быть независима от требований других. Порты позволяют разделять интерфейсы компонента на дискретные пакеты и использовать обособленно. Инкапсуляция и независимость, обеспечиваемая портами, повышают степень заменяемости компонента.

Порт схематически представлен маленьким квадратом на боковой грани компонента – это отверстие в границе инкапсуляции компонента. Как предоставляемый, так и требуемый интерфейс может быть соединен с символом порта. Предоставляемый интерфейс изображает сервис, который может быть запрошен извне через данный порт, а требуемый интерфейс – сервис, который порт должен получить от какого-либо другого компонента. У каждого порта есть *имя*, а следовательно, он может быть идентифицирован по компоненту и имени. Последнее могут использовать внутренние части компонента для идентификации порта, через который следует отправлять и получать сообщения. Имя компонента вместе с именем порта идентифицирует порт для использования его другими компонентами.

Части также могут иметь множественность, поэтому одна часть компонента может соответствовать нескольким экземплярам в пределах одного экземпляра компонента.

Порты – это часть компонента. Экземпляры портов создаются и уничтожаются вместе с экземпляром компонента, которому они принадлежат. Порты также могут иметь множественность; это означает возможность существования нескольких экземпляров порта внутри экземпляра компонента. Каждый порт компонента имеет соответствующий массив экземпляров. Хотя все экземпляры портов в массиве удовлетворяют одному и тому же интерфейсу и принимают запросы одних и тех же видов, они могут находиться

в различных состояниях и иметь разные значения данных. Например, каждый экземпляр в массиве может иметь свой уровень приоритета (экземпляр порта с наибольшим уровнем приоритета обслуживается первым).

На рис. 15.2 представлена модель компонента `Ticket Seller` (Продавец билетов) с портами. У каждого порта есть имя и необязательный тип, показывающий, каково назначение данного порта. Компонент имеет порты для продажи билетов, объявлений и обслуживания кредитных карт.

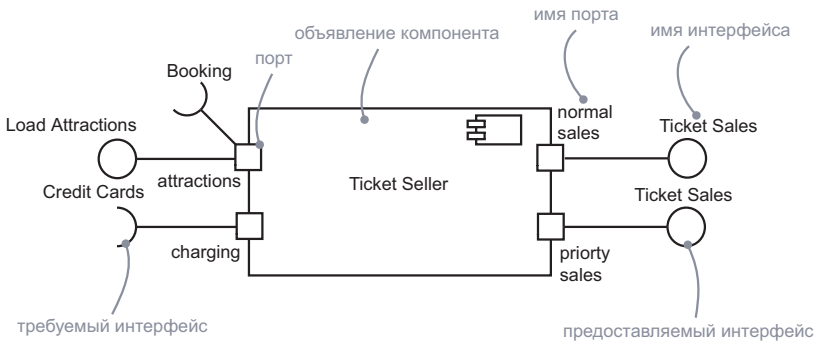


Рис. 15.2. Порты компонента

Есть два порта для продажи – один для обычных покупателей и один для привилегированных. Оба предоставляют один и тот же интерфейс типа `Ticket Sales` (Продажа билетов). Порт обслуживания кредитных карт имеет требуемый интерфейс; любой компонент, который его предоставляет, может удовлетворить его. Порт объявлений имеет как предоставляемый, так и требуемый интерфейсы. Используя интерфейс `Load Attractions` (Информация о развлечениях), театр может передавать афиши и другую информацию о спектаклях в базу данных, используемую для продажи билетов. При помощи интерфейса `Booking` (Заказ) компонент – продавец билетов может запрашивать у театров сведения о наличии билетов и приобретать их.

Внутренняя структура

Компонент может быть реализован как единый фрагмент кода, но в больших системах желательно иметь возможность строить крупные компоненты из малых, которые используются в качестве строительных блоков. Внутренняя структура компонента содержит части, которые вкупе с соединениями между ними составляют его реализацию. Во многих случаях внутренние части могут быть

экземплярами более мелких компонентов, связанных статически через порты для обеспечения необходимого поведения, без необходимости для автора модели специфицировать дополнительную логику.

Часть – это единица реализации компонента, которой присвоены имя и тип. В экземпляре компонента содержится по одному или по несколько экземпляров каждой части определенного типа. Часть имеет множественность в пределах компонента. Если эта множественность больше единицы, то в экземпляре компонента может быть ряд экземпляров компонента данного типа. Если множественность представлена не одним целым числом, то количество экземпляров части может варьироваться в разных экземплярах компонента. Экземпляр компонента создается с минимальным количеством частей (прочие при необходимости добавляются позднее). Атрибут класса – это разновидность части: он имеет тип и множественность, и у каждого экземпляра класса есть один или несколько экземпляров атрибута данного типа.

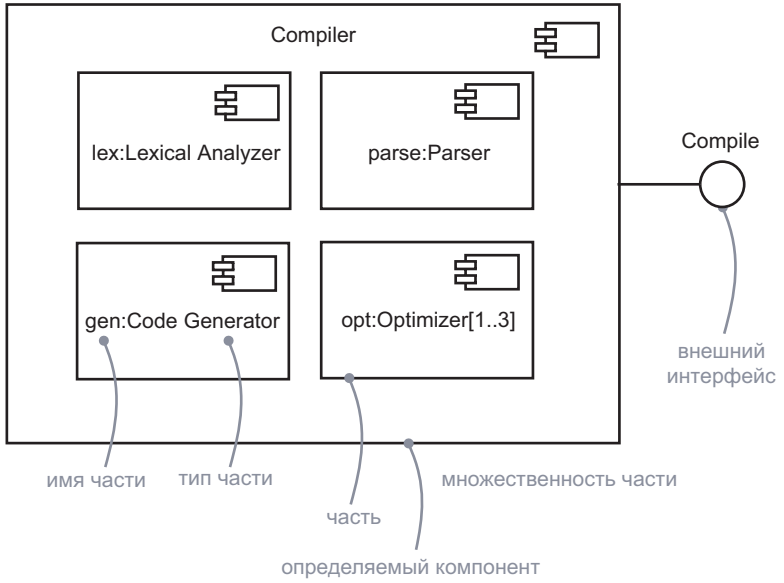


Рис. 15.3. Части компонента

На рис. 15.3 показан компонент-компилятор, состоящий из частей четырех видов. В их числе – лексический анализатор, синтаксический анализатор (parser), генератор кода и от одного до трех оптимизаторов. Более полная версия компилятора может быть сконфигурирована с разными уровнями оптимизации; в данной версии необходимый оптимизатор может выбираться во время исполнения.

Отметим, что часть – это не то же самое, что класс. Каждая часть идентифицируется по ее имени так же, как в классе различаются атрибуты. Допустимо наличие нескольких частей одного и того же типа, но вы можете различать их по именам и они предположительно выполняют разные функции внутри компонента. Например, компонент `AirTicket Sales` (Продажа авиабилетов) на рис. 15.4 может включать отдельные части `Sales` для постоянных и обычных клиентов; обе они работают одинаково, но первая обслуживает только привилегированных клиентов, дает больше шансов избежать очередей и предоставляет некоторые льготы. Поскольку это компоненты одинакового типа, их потребуется различать по именам. Другие два компонента типов `SeatAssignment` (УказаниеМест) и `InventoryManagement` (УправлениеСписками) не требуют имен, поскольку присутствуют в одном экземпляре внутри компонента `Air Ticket Sales`.

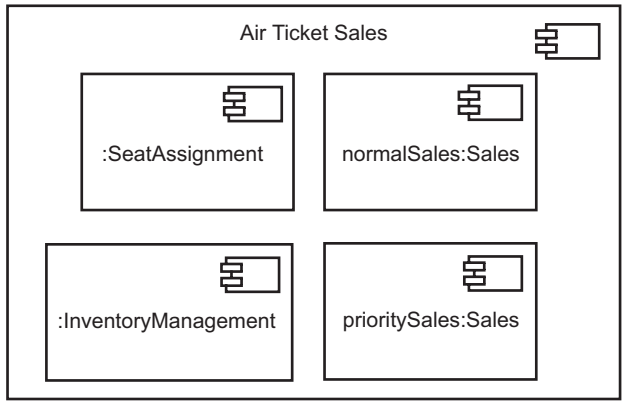


Рис. 15.4. Части одного типа

Если части представляют собой компоненты с портами, вы можете связать их друг с другом через эти порты. Два порта могут быть подключены друг к другу, если один из них предоставляет данный интерфейс, а другой требует его. Подключение портов подразумевает, что для получения сервиса требующий порт вызовет предоставляющий . Преимущества портов и интерфейсов в том, что кроме них больше ничего не важно; если интерфейсы совместимы, то порты могут быть подключены друг к другу. Инструментальные средства способны автоматически генерировать код вызова одного компонента от другого. Также их можно подключить к другим компонентам, предоставляющим те же интерфейсы, когда таковые появятся и станут доступны. «Проводок» между двумя портами называется *коннектором*. В экземпляре охватывающего компонента он представляет просто ссылку (link) или временную ссылку (transient

link). Простая ссылка – это экземпляр обычной ассоциации. Временная ссылка представляет связь использования между двумя компонентами. Вместо обычной ассоциации она может быть обеспечена параметром процедуры или локальной переменной, которая служит целью операции. Преимущество портов и интерфейсов в том, что эти два компонента не обязаны ничего «знать» друг о друге на этапе дизайна, до тех пор пока совместимы их интерфейсы.

Коннекторы изображаются двумя способами (рис. 15.5). Если два компонента явно связаны друг с другом (либо напрямую, либо через порты), достаточно провести линию между ними или их портами. Если же два компонента подключены друг к другу, потому что имеют совместимые интерфейсы, вы можете использовать нотацию «шарик–гнездо», чтобы показать, что между этими компонентами не существует постоянной связи, хотя они и соединены внутри объемлющего компонента. Вы в любое время подставите вместо каждого из них любой другой компонент, если он удовлетворяет интерфейсу.

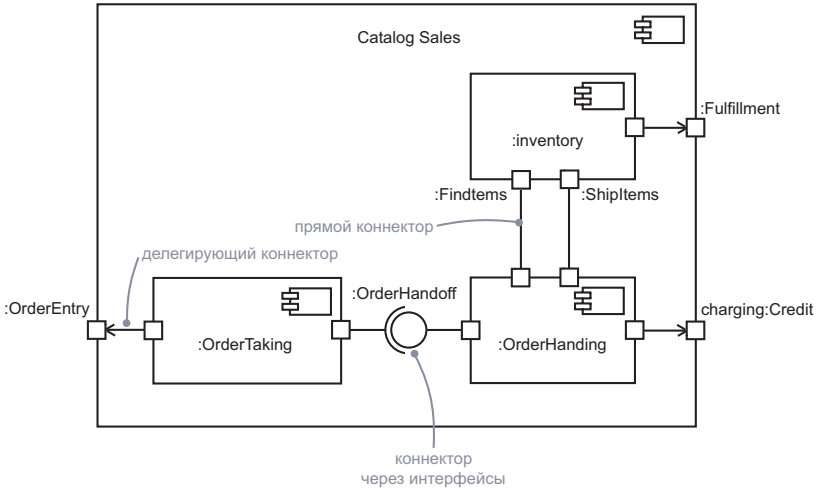


Рис. 15.5. Коннекторы

Также можно связать внутренние порты с внешними портами объемлющего компонента. В таком случае речь идет о *делегирующем коннекторе*, поскольку сообщения из внешнего порта делегируются внутреннему. Подобная связь изображается стрелочкой, направленной от внутреннего порта к внешнему. Вы можете представить ее двояко, как вам больше нравится. Во-первых, можно считать, что внутренний порт – то же самое, что и внешний; он вынесен на границу и допускает подключение извне. Во-вторых, примите

во внимание, что любое сообщение, пришедшее на внешний порт, немедленно передается на внутренний, и наоборот. Это неважно – поведение будет одинаковым в любом случае.

В примере на рис. 15.5 показано использование внутренних портов и разных видов коннекторов. Внешние запросы, приходящие на порт OrderEntry (ПередачаЗаказа), делегируются на внутренний порт подкомпонента OrderTaking (ПриемЗаказа). Этот компонент, в свою очередь, отправляет свой вывод на свой порт OrderHandoff (ПереводЗаказа). Последний подключен по схеме «шарик-гнездо» к подкомпоненту OrderHandling (УправлениеЗаказами). Данный вид подключения предполагает, что у компонентов не существует знаний друг о друге; вывод может быть подсоединен к любому другому компоненту, который соответствует интерфейсу OrderHandOff. Компонент OrderHandling взаимодействует с компонентом Inventory (Опись) для поиска элементов на складе. Это взаимодействие выражено прямым коннектором; поскольку никаких интерфейсов не показано, можно предположить, что данное подключение более плотно, то есть обеспечивает более сильную связь. Как только элемент найден на складе, компонент OrderHandling обращается к внешнему сервису Credit (Кредит) – об этом свидетельствует делегирующий коннектор к внешнему порту changing (изменение).

Диаграммы взаимодействия обсуждаются в главе 19.

Как только внешний сервис Credit дает ответ, компонент OrderHandling связывается с другим портом ShipItems (ДеталиДоставки) компонента Inventory, чтобы подготовить пересылку заказа. Компонент Inventory обращается к внешнему сервису Fullfillment (Исполнение) для осуществления доставки.

Отметим, что диаграмма компонентов показывает структуру и возможные способы доставки сообщений компонента. Последовательность сообщений в компоненте она, однако, не отражает. Последовательности и другие виды динамической информации могут быть представлены на диаграмме взаимодействия.

На заметку. Внутренняя структура, включая порты, части и коннекторы, может быть использована как реализация любых классов – не обязательно компонентов. На самом деле нет большой разницы в семантике между классами и компонентами. Однако зачастую удобно применять соглашение о том, что компоненты используются для инкапсуляции концепций, имеющих внутреннюю структуру (в частности, таких, которые не отображаются непосредственно на отдельный класс в реализации).

Типичные приемы моделирования

Моделирование структурированных классов

Структурированный класс может быть использован для моделирования структур данных, в которых части имеют контекстно-зависимые соединения, существующие только в пределах класса. Обычные атрибуты или ассоциации могут определять составные части класса, но части не могут быть связаны друг с другом на простой диаграмме классов. Класс, внутренняя структура которого показана с помощью частей и коннекторов, позволяет избежать этой проблемы.

Чтобы смоделировать структурированный класс, необходимо:

- Идентифицировать внутренние части класса и их типы.
- Дать каждой части имя, отражающее ее назначение в структурированном классе, а не ее тип.
- Нарисовать коннекторы между частями, которые обеспечивают коммуникацию или имеют контекстно-зависимые связи.
- Не стесняться использовать другие структурированные классы как типы частей, но помнить, что подключение к частям нельзя осуществлять внутри другого структурированного класса – можно подключаться только к их внешним портам.

Рис. 15.6 демонстрирует дизайн структурированного класса TicketOrder (ЗаказБилетов). Этот класс имеет четыре части и один обычный атрибут price (цена). Далее, customer (покупатель) – это объект Person (Человек); он может иметь или не иметь статус priority (приоритет), поэтому часть priority показана с множественностью 0..1. Коннектор от customer к priority имеет ту же множественность. Поскольку каждый клиент вправе бронировать одно или несколько посадочных мест, у объекта seat (место) тоже есть значение множественности. Нет необходимости показывать коннектор от customer к seats, потому что эти объекты и так находятся в одном структурированном классе. Отметим, что класс Attraction (Категория) обведен пунктирной рамкой. Это значит, что данная часть представляет собой ссылку на объект, которым не владеет структурированный класс. Ссылка создается и уничтожается вместе с экземпляром класса TicketOrder, но экземпляры Attraction независимы от класса TicketOrder. Часть seat подключена к ссылке attraction, поскольку заказ может включать посадочные места разных категорий и каждое бронирование билета должно быть сопоставлено определенной категории посадочного места. Мы видим по значению множественности, что каждое резервирование Seat подключено строго к одному объекту Attraction.

Моделирование программного интерфейса API

Если вы разрабатываете систему, состоящую из частей-компонентов, вам часто требуется видеть интерфейсы прикладного программирования (application programming interfaces, API), посредством которых эти части связываются друг с другом. API представляют программные соединения в системе, которые можно смоделировать, используя интерфейсы и компоненты.

По сути, API – это интерфейс, который реализуется одним или несколькими компонентами. Как разработчик, вы на самом деле заботитесь только о самом интерфейсе – какие именно компоненты реализуют операции интерфейса, неважно до тех пор, пока какой-нибудь компонент реализует их. С точки зрения управления конфигурацией системы, однако, эти реализации важны, потому что вам необходимо быть уверенным в том, что когда вы опубликуете API, то будет доступна некая соответствующая ему реализация. К счастью, с помощью UML можно смоделировать оба представления.

Операции, ассоциированные с любым семантически насыщенным API, будут встречаться достаточно часто, потому в большинстве случаев вам не понадобится визуализировать все их сразу. Напротив, вы будете стремиться к тому, чтобы оставлять операции на периферии ваших моделей и использовать интерфейсы в качестве дескрипторов, посредством которых можно будет найти все эти наборы операций. Если вы хотите конструировать исполнимые системы на основе таких API, то вам понадобится детализировать модели настолько, чтобы инструменты разработки были способны производить компиляцию в соответствии со свойствами интерфейсов. Наряду с сигнатурой каждой операции вы, возможно, захотите отобразить варианты использования, объясняющие, как нужно применять каждый интерфейс.

Чтобы смоделировать API, необходимо:

- Идентифицировать соединения в системе и смоделировать каждое из них в виде интерфейса, собирая вместе образующие его атрибуты и операции.
- Показать только те свойства интерфейса, которые важны для визуализации в данном контексте; в противном случае скрыть их, сохраняя только в спецификации интерфейса для последующей ссылки (по мере необходимости).
- Смоделировать реализацию каждого API лишь в той мере, в которой она важна для показа конкретной конфигурации.

На рис. 15.7 представлен API компонента анимации. Вы видите здесь четыре интерфейса, составляющие API: `IApplication` (IПриложение), `IModels` (IМодели), `IRendering` (IВизуализация) и `IScripts` (IСценарии). Другие компоненты могут использовать один или несколько таких интерфейсов по необходимости.

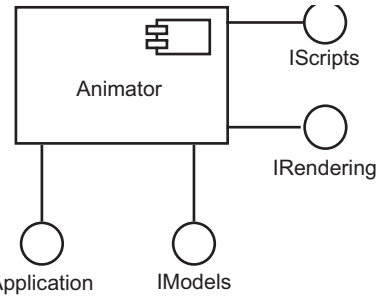


Рис. 15.7. Моделирование API

Советы и подсказки

Компоненты позволяют вам инкапсулировать части вашей системы, чтобы уменьшить количество зависимостей, сделать их явными, а также повысить взаимозаменяемость и гибкость на случай, если система должна будет изменяться в будущем. Хороший компонент наделен следующими характеристиками:

- инкапсулирует сервис с хорошо очерченным интерфейсом и границами;
- имеет внутреннюю структуру, которая допускает возможность ее описания;
- не комбинирует несвязанной функциональности в пределах одной единицы;
- организует внешнее поведение, используя интерфейсы и порты в небольшом количестве;
- взаимодействует только через объявленные порты.

Если вы решили показать реализацию компонента, используя вложенные подкомпоненты, примите во внимание следующее:

- не злоупотребляйте использованием подкомпонентов. Если их слишком много, чтобы они легко уместились на одной странице, применяйте дополнительные уровни декомпозиции некоторых из них;
- убедитесь, что подкомпоненты взаимодействуют только через определенные порты и коннекторы;
- определите, какие подкомпоненты непосредственно взаимодействуют с внешним миром, и моделируйте их делегирующими коннекторами.

Когда вы изображаете компонент в UML:

- дайте ему имя, которое ясно описывает его назначение. Таким же образом именуйте интерфейсы;
- присваивайте имена подкомпонентам и портам в случае, если их значение нельзя определить по их типам или же в модели присутствует множество частей одного типа;
- скрывайте ненужные детали. Вы не должны показывать все детали реализации на диаграмме компонентов;
- показывайте динамику компонентов, используя диагр аммы взаимодействия.

Часть IV

Основы моделирования поведения

Глава 16. Взаимодействия

В этой главе:

- Роли, ссылки, сообщения, действия и последовательности
- Моделирования потоков управления
- Создание хорошо структурированных алгоритмов

В грамотно выстроенной системе объекты не обособленны – они взаимодействуют друг с другом, передавая сообщения. Взаимодействие – это такое поведение, которое предполагает обмен сообщениями между множеством объектов в пределах определенного контекста и предназначено для достижения определенных целей.

Взаимодействия используются для того, чтобы моделировать динамические аспекты кооперации, которые представляют собой сообщества объектов, играющих специфические роли и работающих совместно для обеспечения поведения, большего чем поведение простой суммы элементов. Эти роли представляют прототипные экземпляры классов, интерфейсов, компонентов, узлов и вариантов использования. Их динамические аспекты визуализируются, специфицируются, конструируются и документируются в виде потоков управления. Эти потоки управления могут быть представлены в системе простыми последовательными потоками либо включать ветвление, циклы, рекурсии и параллелизм. Вы можете моделировать взаимодействие двумя способами: выделяя временную последовательность сообщений или же подчеркивая эту последовательность в контексте некоторой структурной организации объектов.

Хорошо структурированные взаимодействия подобны хорошо структурированным алгоритмам: они эффективны, просты, адаптируемы и понятны.

Введение

Здание – это живая сущность. Хотя каждое здание и состоит из статических объектов, таких как кирпичи, раствор, балки, пластик, стекло и сталь, все эти вещи в своей совокупности работают динамически для достижения такого поведения, которое удобно

Разница между строительством небоскреба и собачьей будки обсуждается в главе 1.

жителям дома. Двери и окна открываются и закрываются. Включается и выключается свет. Системы отопления, кондиционирования, терморегуляции и вентиляции сообща обеспечивают температурный режим. В «интеллектуальных» домах сенсорные датчики определяют наличие или отсутствие деятельности жителей и соответствующим образом регулируют освещение, отопление, охлаждение и громкость звука в аудио- и телеаппаратуре. Здания строятся с учетом перемещения людей и вещей с места на место. К тому же дом должен адаптироваться к изменениям температуры, повышающейся и понижающейся в течение суток, а также к ее сезонным колебаниям. Все хорошо структурированные здания проектируются с учетом противостояния динамическим воздействиям: ветру, землетрясениям и перемещению жителей.

Моделирование структурных аспектов системы обсуждается в частях II и III; также вы можете моделировать динамические аспекты системы, используя конечные автоматы, как показано в главе 22. Диаграммы объектов обсуждаются в главе 14, диаграммы взаимодействия – в главе 19, кооперации – в главе 28.

То же самое касается программных систем. Скажем, система управления полетами может управлять многими терабайтами информации, которая большую часть времени хранится нетронутой на некотором диске и становится необходимой лишь в результате некоторых обстоятельств, например в связи с продажей билетов, вылетом самолета или планированием расписания полетов. В реактивных системах, например в компьютерах и микроволновых печах, объекты вызываются к жизни и выполняют свою работу, когда систему приводит в действие определенное событие – нажатие клавиши или истечение заданного срока.

В UML статические аспекты системы моделируются такими элементами, как диаграммы классов и диаграммы объектов. Эти диаграммы позволяют визуализировать, специфицировать, конструировать и документировать сущности, которые составляют систему, – в частности классы, интерфейсы, компоненты, узлы, варианты использования и их экземпляры вместе со связями, существующими между ними.

Динамические аспекты системы в UML моделируются *взаимодействиями*. Как и диаграммы объектов, взаимодействия статически определяют фазы поведения, представляя все объекты, ведущие совместную работу для достижения определенной цели. Однако, в отличие от диаграмм объектов, взаимодействия также представляют сообщения, передаваемые от объекта к объекту. Чаще всего сообщения вызывают операции или передачу сигнала; кроме того, могут сопровождаться созданием и уничтожением других объектов.

Взаимодействия используются для моделирования потока управления в пределах операции, класса, компонента, варианта использования или системы в целом. С помощью диаграмм взаимодействия вы можете составить представление об этих потоках двумя способами. Во-первых, можно сосредоточиться на том, как сообщения передаются во времени, а во-вторых – на структурных связях между объектами во взаимодействии, и затем рассмотреть, как сообщения передаются в контексте структуры.

Графическое представление сообщений в UML проиллюстрировано на рис. 16.1. Эта нотация позволяет вам отразить наиболее важные свойства сообщения: имя, параметры (если таковые есть) и последовательность. Графически сообщение изображается линией со стрелкой и почти всегда включает имя его операции.

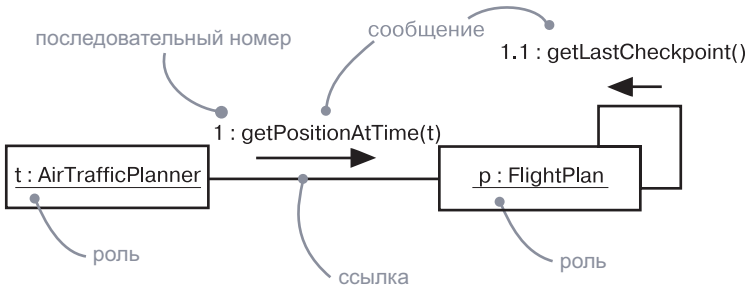


Рис. 16.1. Сообщения, ссылки и последовательность

Базовые понятия

Диаграммы внутренней структуры показывают структурные соединения между ролями, как показано в главе 15; объекты обсуждаются в главе 14, системы и подсистемы – в главе 32, кооперации – в главе 28.

Взаимодействие (interaction) – это поведение, в которое вовлечено множество сообщений, передаваемых между объектами, которые играют определенные роли в некоем контексте для достижения заданной цели.

Сообщение – это спецификация взаимодействия объектов, которая передает информацию и ожидает последующих действий.

Контекст

Взаимодействие имеет место всякий раз, когда объекты соединены друг с другом. Вы найдете взаимодействия в кооперациях объектов, существующих в контексте системы или подсистемы, а также в контексте операций и, наконец, классов.

Чаще всего взаимодействия встречаются в кооперациях объектов, представленных в контексте системы или подсистемы в целом. Например, в системе Web-коммерции вы обнаружите такие клиентские объекты, как экземпляры классов `BookOrder` (ЗаказКниги) и `OrderForm` (БланкЗаказа), которые взаимодействуют друг с другом. Там же присутствуют клиенты (опять-таки экземпляры `BookOrder`), взаимодействующие с объектами на сервере, например экземплярами `BockOrderManager` (МенеджерЗаказов). Таким образом, взаимодействия не только включают локализованную кооперацию объектов (как, например, вокруг `OrderForm`), но и распространяются через многие концептуальные уровни системы (в частности, взаимодействия с `BackOrderManager`).

Операции обсуждаются в главах 4 и 9; моделирование операций рассматривается в главах 20 и 28.

Помимо прочего, вы найдете взаимодействия между объектами в реализации операций. Параметры операций, их локальные переменные, глобальные по отношению к операции (но видимые в ней) объекты могут взаимодействовать друг с другом для обеспечения работы алгоритма, реализующего операцию. Например, операция `moveToPosition(p : Position)` (перейтиНаПозицию (p : Позиция), определенная в классе движущегося робота, включает взаимодействие с параметром `p`, глобальными по отношению к операции объектами, такими как `CurrentPosition` (ТекущаяПозиция), и, возможно, несколькими локальными объектами: локальными переменными, используемыми операцией для вычисления промежуточных точек на пути к новой позиции.

Классы обсуждаются в главах 4 и 9.

Наконец, взаимодействия в контексте класса можно применять для визуализации, специфицирования, конструирования и документирования его семантики. Например, чтобы понять смысл класса `RayTraceAgent`, можно описать взаимодействие, которое покажет, как его атрибуты кооперируются друг с другом (и с глобальными по отношению к экземплярам класса объектами, и с параметрами, определенными для его операций).

Компоненты обсуждаются в главе 15, узлы – в главе 27, варианты использования – в главе 17; моделирование реализаций вариантов использования рассматривается в главе 28, классификаторы – в главе 9.

На заметку. Взаимодействие можно обнаружить и в представлении компонента, узла или варианта использования, которые по своей сути являются разновидностями классификаторов UML. В контексте варианта использования взаимодействие описывает сценарий, представляющий, в свою очередь, отдельный поток варианта использования.

Объекты и роли

Объекты, которые участвуют во взаимодействии, могут быть либо конкретными сущностями, либо прототипами. Как конкретные сущности они выражают вполне определенные реалии. Например, `p` – экземпляр класса `Person` (Человек) – описывает конкретное лицо. С другой стороны, как прототипная сущность `p` может представлять любой экземпляр класса `Person`.

На заметку. В кооперациях взаимодействия часто связывают не объекты реального мира, а прототипные сущности, играющие определенные роли, хотя иногда бывает удобно описать кооперации и между конкретными объектами.

В контексте взаимодействия вы можете обнаружить экземпляры классов, компонентов, узлов и вариантов использования. Хотя абстрактные классы и интерфейсы по определению не могут иметь прямых экземпляров, их экземпляры можно представить во взаимодействии. Последние не являются прямыми экземплярами абстрактных классов или интерфейсов, но могут представлять, соответственно, косвенные, или прототипные, экземпляры любых конкретных потомков абстрактного класса либо некоего конкретного класса, реализующего интерфейс.

Экземпляры обсуждаются в главе 13, диаграммы объектов – в главе 14.

Вы можете рассматривать диаграмму объектов как представление статического аспекта взаимодействия, устанавливающее стадию взаимодействия посредством специфицирования всех объектов, которые работают вместе. Между тем взаимодействие идет еще дальше, представляя динамическую последовательность сообщений, которые могут передаваться по ссылкам, соединяющим данные объекты.

Ссылки и коннекторы

Ассоциации обсуждаются в главах 5 и 10, коннекторы и роли – в главе 15.

Ссылка (link) – это семантическое соединение объектов друг с другом. Вообще говоря, ссылка – это экземпляр ассоциации. Как показывает рис. 16.2, когда между классами установлена связь ассоциации, между экземплярами этих классов обязательно должна существовать ссылка. При наличии ссылки между двумя объектами один из них может посылать сообщения другому.

Ссылка указывает путь, по которому один объект может посылать сообщения другому (или самому себе). В большинстве случаев достаточно констатировать сам факт, что такой путь существует. Если вам нужно точнее описать его, вы можете снабдить соответствующий его конец одним из нижеперечисленных ограничений:

- `association` – указывает, что соответствующий объект виден ассоциации;
- `self` – указывает, что соответствующий объект виден, поскольку он является диспетчером операции;
- `global` – указывает, что соответствующий объект виден, поскольку он находится в объемлющей области действия;
- `local` – указывает, что соответствующий объект виден, поскольку он находится в локальной области действия;
- `parameter` – указывает, что соответствующий объект виден, потому что он является параметром.

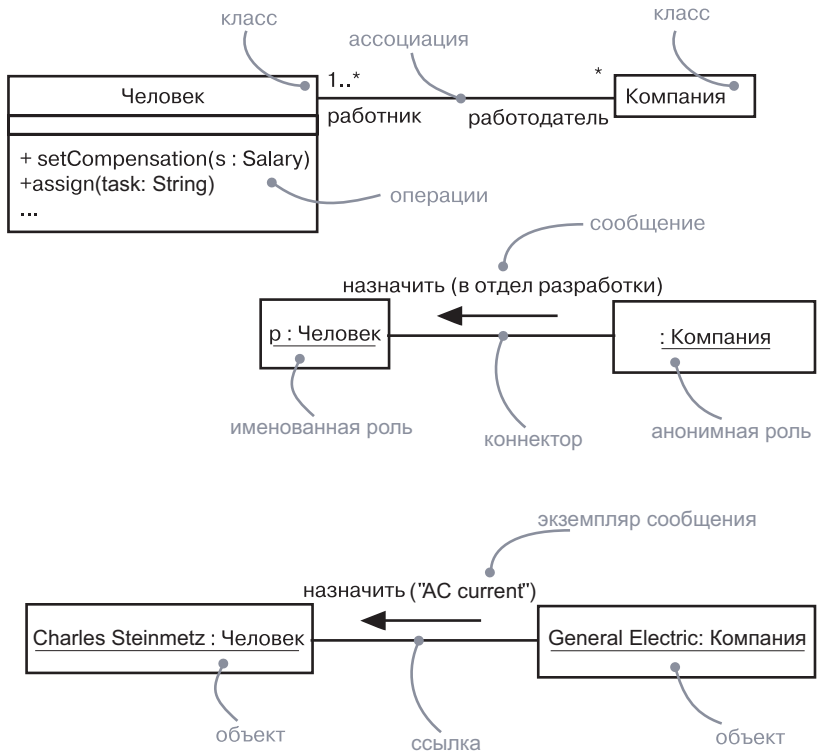


Рис. 16.2. Ассоциации, ссылки и коннекторы

На заметку. Как экземпляр ассоциации ссылка может изображаться вместе с большинством дополнений, соответствующих ассоциации, таких как его имя, имя ассоциированной роли, навигация и агрегация. Однако множественность неприменима к ссылкам, поскольку они являются экземплярами ассоциаций.

Роли, коннекторы и внутренние структуры обсуждаются в главе 15, взаимодействия – в главе 28.

Разработчики большинства моделей более заинтересованы в прототипных объектах и ссылках, существующих в определенном контексте, чем в конкретных индивидуальных объектах с их ссылками. Прототипный объект называется *ролью*, а прототипная связь – *коннектором*; контекст – это кооперация или внутренняя структура классификатора. Множественность ролей и коннекторов определена относительно включающего их контекста. Например, множественность роли равная 1 означает, что на один объект, представляющий контекст, приходится один объект, представляющий роль. Кооперация

или внутренняя структура может использоваться много раз, как и объявление класса, причем каждое такое использование связано с отдельным набором объектов и ссылок для данного контекста ролей и ссылок.

В качестве примера в верхней части рис. 16.2 показана диаграмма классов, которая объявляет классы `Person` (Человек) и `Company` (Компания), а также существующую между ними ассоциацию «многие-ко-многим» `employee–employer` (наниматель–работник). Средняя часть рисунка представляет содержимое кооперации `WorkAssignment` (НазначениеНаДолжность), название которой говорит само за себя. Она включает в себя две роли с коннектором между ними. В нижней части рисунка вы видите экземпляр кооперации, в котором есть объекты и ссылки, представляющие роли и коннекторы. Конкретное сообщение представляет прототипное объявление сообщения в данной кооперации.

Сообщения

Диаграммы объектов обсуждаются в главе 14.

Предположим, у вас есть набор объектов и ряд ссылок, соединяющих эти объекты. Если это все, чем вы располагаете, то речь идет о полностью статической модели, описываемой диаграммой объектов. Последняя моделирует состояние сообщества объектов в заданный момент времени и применяется в случае, когда нужно визуализировать, специфицировать, конструировать или документировать статическую структуру объектов.

Операции обсуждаются в главах 4 и 9, события – в главе 21, экземпляры – в главе 13.

Теперь допустим, что вы хотите смоделировать изменяющееся состояние сообщества объектов за некоторый период времени. Чтобы представить это наглядно, вообразим фильм, каждый кадр которого демонстрирует перемещение набора объектов в следующий момент времени. Если эти объекты не являются статическими, то вы увидите передачу сообщений другим объектам, отправку событий и вызов операций. Вдобавок в каждом кадре можно явно визуализировать текущее состояние и роль индивидуальных экземпляров.

Сообщение – это спецификация взаимодействия объектов, обеспечивающего передачу информации, которая должна инициировать определенные действия. Момент приема экземпляра сообщения может рассматриваться как случай возникновения события; *случай* (occurrence) – это термин UML, описывающий экземпляр события.

Когда вы передаете сообщение, то в результате его приема обычно происходит некоторое действие. Оно может заключаться в изменении состояния целевого объекта и объектов, доступных ему.

В UML можно моделировать несколько типов сообщений.

Операции обсуждаются в главах 4 и 9, сигналы – в главе 21.

- `call` (вызвать) – вызывает операцию объекта. Объект может посылать сообщение самому себе, в результате происходит локальный вызов операции;
- `return` (возвратить) – возвращает значение вызывающему объекту;
- `send` (послать) – посылает сигнал объекту;
- `create` (создать) – создает объект;
- `destroy` (уничтожить) – уничтожает объект. Объект может уничтожить сам себя.

На заметку. В дополнение к пяти основным видам сообщений вы можете производить и более сложные действия над индивидуальными объектами. UML не специфицирует синтаксис или семантику таких действий; предполагается, что инструментальные средства должны использовать разные языки описания действий либо применять синтаксис языков программирования.

Создание и удаление объектов визуализируются как стереотипы, описание которых приводится в главе 6. Разница между синхронными и асинхронными сообщениями наиболее существенна в контексте параллельности; об этом пойдет речь в главе 23.

В UML эти типы сообщений различаются визуально, как показано на рис. 16.3.

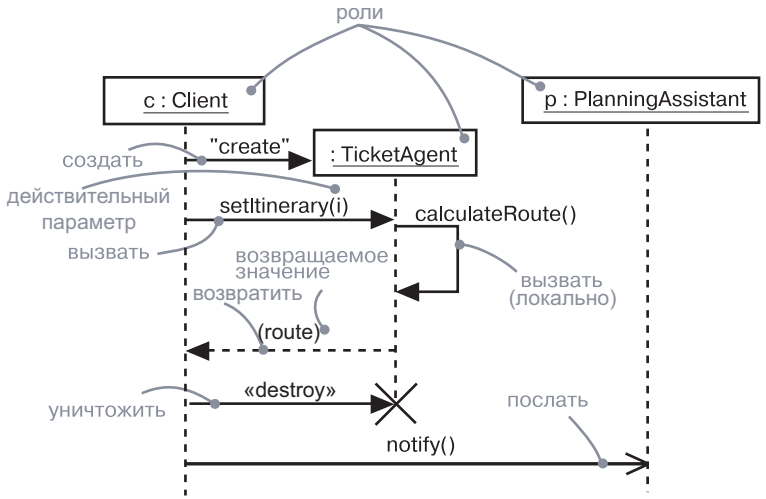


Рис. 16.3. Сообщения

Из всех видов сообщений наиболее часто вам придется моделировать вызов (когда один объект вызывает операцию другого объекта или свою собственную). Объект не может вызвать любую

Классы обсуждаются в главах 4 и 9.

произвольную операцию. Если объект наподобие c из предыдущего примера вызывает, скажем, операцию setItinerary (указатьНаправление) на экземпляре класса TicketAgent (АгентПоПродажеБилетов) – то есть она должна быть объявлена в классе TicketAgent или в одном из его родителей, – то эта операция должна быть видима для вызывающего объекта c.

На заметку. Такие языки, как C++, типизированы статически (хотя и полиморфно). Это означает, что корректность вызовов проверяется во время компиляции. А вот языки наподобие SmallTalk являются динамически типизированными – следовательно, до времени исполнения нельзя определить, может ли объект принять конкретное сообщение. В UML хорошо согласованная модель вообще может быть проверена статически с помощью инструментальных средств, поскольку на момент моделирования проектировщик, как правило, представляет себе назначение операций.

Когда объект вызывает операцию или посылает сигнал другому объекту, вы можете снабдить сообщение конкретным параметром. Аналогичным образом, когда один объект возвращает управление другому, вы также можете смоделировать возвращаемое значение.

Интерфейсы обсуждаются в главе 11.

Сообщения могут иметь отношение и к посылке сигналов. *Сигнал* (signal) – это значение объекта, передаваемое целевому объекту асинхронно. После отправки сигнала объект, который его отправил, продолжает свою деятельность.

На заметку. Вы также можете определить операцию по классу или интерфейсу, в котором она объявлена. Например, вызов операции register (зарегистрировать) на экземпляре Student (Студент) может полиморфно вызвать любую операцию с этим именем в иерархии классов Student; вызов Imember::register (Iучастник::зарегистрировать) вызывает операцию, специфицированную в интерфейсе Imember и реализованную неким соответствующим классом, также находящимся в иерархии Student.

Получая сообщение сигнала, целевой объект независимо принимает решение о том, что с ним нужно делать. Обычно сигналы инициируют переход автомата целевого объекта в другое состояние. Инициализация такого перехода заставляет целевой объект выполнить некие действия и изменить свое состояние. В системах с асинхронной передачей событий взаимодействующие объекты исполняются параллельно и независимо. Они разделяют информацию только посредством передачи сообщений, поэтому не возникает опасности конфликта за разделяемую память.

Последовательности

Процессы и потоки обсуждаются в главе 23.

Когда один объект передает сообщение другому (по сути, делегируя ему некоторое действие), то принимающий объект может, в свою очередь, послать сообщение еще одному объекту, а тот – следующему, и т.д. Этот поток сообщений формирует *последовательность* (sequencing). Всякая последовательность должна иметь начало; ее запуск выполняется определенным процессом или потоком. Более того, последовательность продолжается до тех пор, пока существует владеющий ею процесс или поток. Системы «нон-стоп» (работающие без остановки),которые, в частности, встречаются в приложениях управления реального времени, продолжают работать до тех пор, пока работает узел, на котором они запущены.

Системы обсуждаются в главе 32.

Каждый процесс и поток в системе определяет отдельный поток управления, в каждом из которых сообщения упорядочены во времени. Чтобы лучше визуализировать последовательность сообщений, вы можете явно смоделировать их порядок относительно начала последовательности, снабдив сообщение префиксом, указывающим его номер с двоеточием в качестве разделителя.

Диаграмма коммуникации отражает поток сообщений между ролями в пределах кооперации. Поток сообщений передается через соединения, существующие в кооперациях, как показано на рис. 16.4.

Чаще всего вы можете специфицировать *процедурный* или *вложенный* поток управления, изобразив его в виде сплошной закрашенной стрелки, как на рис. 16.4. В этом случае сообщение findAt (найтиВ) специфицируется как первое сообщение, вложенное во второе в последовательности (2.1).

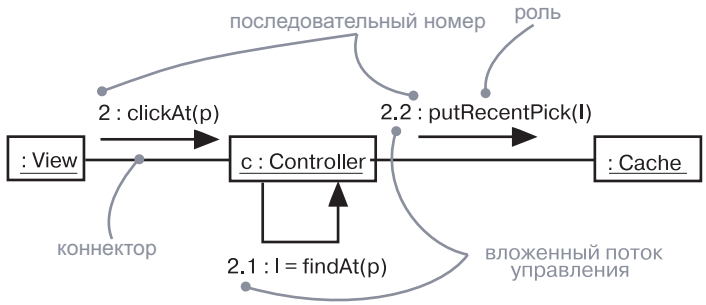


Рис. 16.4. Процедурная последовательность

Реже, но все-таки приходится специфицировать плоский поток управления (он представлен в виде заостренной стрелки), чтобы смоделировать последовательность управления шаг за шагом (см. рис. 16.5). В этом случае сообщение assertCall (принятьЗвонок) указывается как второе в последовательности.

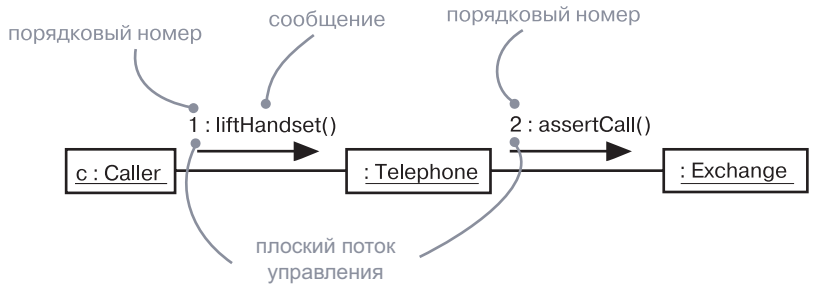


Рис. 16.5. Плоская последовательность

На заметку. Разница между асинхронными и процедурными последовательностями важна в современном мире параллельных вычислений. Чтобы показать общее поведение системы параллельных объектов, используйте асинхронную передачу сообщений. Это самый общий случай. Когда вызывающий объект в состоянии выполнить запрос и дождаться ответа, вы можете использовать процедурный поток управления. Процедурный поток знаком вам из традиционных языков программирования, но надо иметь в виду, что серия вложенных вызовов процедур приводит к появлению стека блокированных объектов, которые временно не в состоянии ничего делать – а это не очень удобно, если речь идет о серверах или разделяемых ресурсах.

Процессы и потоки обсуждаются в главе 23. Вы можете специфицировать асинхронный поток управления, изображаемый в виде стрелочки с одной «веточкой» (см. ту же главу).

При моделировании взаимодействий, которые включают множество потоков управления, особенно важно идентифицировать процесс или поток, который отправил конкретное сообщение. В UML можно отличить один поток управления от другого, снабдив последовательный номер сообщения префиксом, в качестве которого выступает имя процесса или потока, стоящего в начале последовательности. Например, выражение

```
D5 : ejectHatch(3)
```

указывает, что операция `ejectHatch` (открытьЗатвор) отправляется с действительным аргументом 3 как пятое сообщение последовательности, запущенной в процессе или потоке с именем `D`.

Вы можете показать не только действительные аргументы, переданные операции или сигналу в контексте взаимодействия, но и возвращаемые значения функции. Как явствует из нижеследующего выражения, значение `p`, возвращаемое из операции `find`, отправляется с действительным параметром `"Rachelle"`. Это вложенная последовательность: операция выполняется в результате второго сообщения, вложенного в третье, которое, в свою очередь, вложено в первое сообщение последовательности. На той же диаграмме `p` можно использовать в качестве действительного параметра других сообщений.

```
1.3.2 : p := find("Rachelle")
```


Итерации, ветвление и защищенные сообщения обсуждаются в главе 19, отметки времени – в главе 24, стереотипы и ограничения – в главе 6.

На заметку. В UML вы можете моделировать и более сложные формы последовательностей, такие как итерации, ветвление, защищенные сообщения. Вдобавок для моделирования временных ограничений (например, в системах реального времени), с последовательностью можно ассоциировать отметки времени. Другие, более экзотические формы сообщений, в частности отказы и тайм-ауты, можно смоделировать, определив соответствующие стереотипы сообщений.

Создание, модификация и уничтожение

Моделируемые вами объекты, участвующие во взаимодействиях, как правило, существуют в течение всего времени взаимодействия. Однако в некоторых взаимодействиях объекты могут создаваться (по сообщению `create`) и уничтожаться (по сообщению `destroy`). То же самое касается и ссылок: связи между объектами могут возникать и исчезать. Чтобы указать, что объект или ссылка появляется и/или исчезает в процессе взаимодействия, вы можете присоединить примечание к его/ее роли в диаграмме коммуникации.

Линии жизни обсуждаются в главе 19.

При взаимодействии объект обычно изменяет значения своих атрибутов, свое состояние или свои роли. Вы можете представить модификации объекта на диаграмме последовательности, указав состояние или значения на линии жизни.

Внутри диаграммы последовательности время жизни, создание и уничтожение объектов или ролей явно изображаются в виде вертикальных участков их линий жизни. На диаграмме коммуникации создание и уничтожение должны обозначаться примечаниями. Используйте диаграммы последовательности, когда важно показать время жизни объектов.

Представление

Когда вы моделируете взаимодействие, то обычно включаете в него как роли (каждая из которых представляет объекты, появляющиеся в экземпляре взаимодействия), так и сообщения (каждое из которых представляет коммуникацию между объектами вместе с некоторым результирующим действием).

Вы можете визуализировать эти роли и сообщения двумя способами: указывая временной порядок сообщений либо структурную организацию ролей, которые отправляют и принимают сообщения. В UML

Диаграммы взаимодействия обсуждаются в главе 19.

первый тип представления называется диаграммой последовательности, а второй – диаграммой коммуникации. И то и другое представляет собой разновидность диаграмм взаимодействия. Помимо этого UML предусматривает и более специализированную разновидность диаграмм взаимодействия, называемую *временной диаграммой*, которая показывает конкретное время передачи сообщений между ролями. Мы не рассматриваем данный тип диаграмм в настоящей книге. Более подробную информацию можно найти в книге «UML» (ее выходные данные вы найдете во введении, в разделе «Цели»).

Диаграммы последовательности и коммуникации похожи в том смысле, что вы можете взять одну из них и преобразовать в другую, хотя часто они передают разную информацию, и тогда не слишком целесообразно переключаться между ними. Есть и некоторые визуальные отличия. Диаграммы последовательности позволяют вам моделировать линию жизни объекта, которая описывает его существование в определенный период времени (возможно, включая создание и уничтожение объекта). Диаграммы коммуникации, в свою очередь, позволяют моделировать структурные ссылки, возникающие между объектами при их взаимодействии.

Типичные приемы моделирования

Моделирование потока управления

Варианты использования обсуждаются в главе 17, образцы и каркасы – в главе 29, классы и операции – в главах 4 и 9, взаимодействия и интерфейсы – в главе 11, компоненты – в главе 15, узлы – в главе 27.

Чаще всего взаимодействия используются для моделирования потока управления, характеризующего поведение системы в целом (включая варианты использования, образцы, механизмы и каркасы), или же поведения класса либо отдельной операции. В то время как классы, интерфейсы, компоненты, узлы и их связи представляют статические аспекты системы, взаимодействия отражают ее динамические аспекты.

Моделируя взаимодействие, вы, по сути, создаете сценарий действий, происходящих в наборе объектов. При этом могут применяться такие методы, как использование CRC-карт, позволяющих исследовать и продумать все, что касается взаимодействий.

Для моделирования потока управления необходимо:

- Установить контекст взаимодействия: система ли это в целом, класс или индивидуальная операция.
- Определить фазу взаимодействия, прояснив, какие объекты играют те или иные роли. Установить начальные характеристики объектов, включая значения атрибутов, состояние и роль. Дать ролям имена.

Вы можете моделировать динамические аспекты системы, используя автоматы (см. главу 22).

- Если модель описывает структурную организацию объектов, идентифицировать соединяющие их ссылки, которые важны для осуществления коммуникаций при взаимодействии. Специфицировать природу этих ссылок, при необходимости используя стандартные стереотипы и ограничения UML.
- В соответствии с временной последовательностью указать сообщения, передаваемые от объекта к объекту. При необходимости выделить разные виды сообщений; описать параметры и возвращаемые значения, чтобы в должной мере детализировать описание взаимодействия.
- Для более подробного изложения материала снабдить каждый объект, представленный в каждый момент времени, информацией о его состоянии и роли.

Диаграммы последовательности обсуждаются в главе 19.

В примере на рис. 16.6 показан набор ролей, взаимодействующих в контексте механизма публикации и подписки (экземпляр образца проектирования observer – «обозреватель»). Здесь представлены три роли: `p` (`StockQuotePublisher` – ИздательТаблицКотировокАкций), `s1` и `s2` (два экземпляра `StockQuoteSubscriber` – ПодписчикТаблицКотировокАкций). Это наглядный пример диаграммы последовательности, которая показывает временной порядок сообщений.

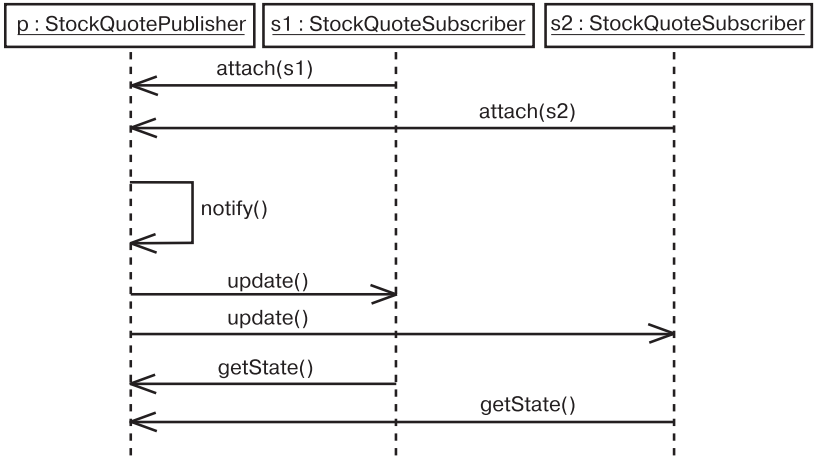


Рис. 16.6. Поток управления во времени

Диаграммы коммуникации обсуждаются в главе 19.

Рис. 16.7 демонстрирует пример, семантически эквивалентный предыдущему, но в виде диаграммы коммуникации, раскрывающей структурную организацию объектов. Здесь показан тот же поток управления, однако в дополнение ко всему прочему визуализированы ссылки между объектами.

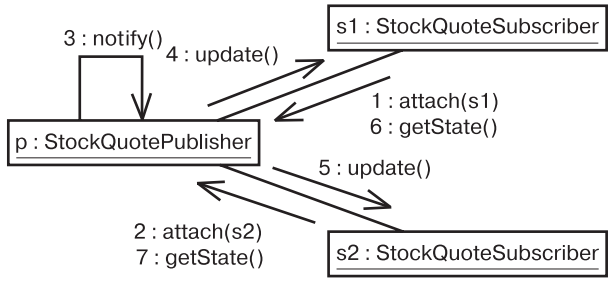


Рис. 16.7. Организация потока управления

Советы и подсказки

Моделируя взаимодействия в UML, помните, что каждое из них представляет динамический аспект сообщества объектов. Хорошо структурированное взаимодействие наделено следующими признаками:

- достаточно просто и описывает только те объекты, которые работают вместе для обеспечения поведения, большего чем поведение суммы этих объектов, взятых порознь;
- имеет четкий контекст и может представлять взаимодействие объектов в контексте операции, класса или системы в целом;
- эффективно обеспечивает нужное поведение с оптимальным балансом времени и ресурсов;
- способно к адаптации: элементы взаимодействия, которые могут изменяться, должны быть изолированы с тем, чтобы их можно было легко модифицировать;
- доступно пониманию и однозначно, тем самым исключая какие бы то ни было шероховатости, скрытые эффекты и неясную семантику.

Когда вы изображаете взаимодействие в UML, учитывайте следующее:

- выберите способ отображения взаимодействия: либо по хронологическому порядку сообщений, либо по их расположению в контексте некоей структурной организации объектов. Применять оба способа одновременно нельзя;
- обратите внимание на то, что события в некоторых субпоследовательностях упорядочены лишь частично. Точнее говоря, каждая из них упорядочена, но относительное время наступления событий в разных субпоследовательностях не фиксировано;

- показывайте только те свойства каждого объекта (значения атрибутов, роли и состояния), которые важны для понимания взаимодействия в его контексте;
- показывайте лишь те свойства каждого сообщения (параметры, семантику параллельности, возвращаемые значения), которые важны для понимания взаимодействия в его контексте.

Глава 17. Варианты использования

В этой главе:

- Варианты использования, действующие лица, включение и расширение
- Моделирование поведения элемента
- Реализация вариантов использования с помощью кооперации

Ни одна грамотно построенная система не существует в изоляции: она взаимодействует с действующими лицами (людьми или системами), которые используют ее для достижения некоторой цели, ожидая от нее определенного поведения. Вариант использования специфицирует это ожидаемое поведение субъекта (системы или ее части), – он описывает последовательности действий, включая их варианты, которые субъект осуществляет для достижения действующим лицом определенного результата.

Варианты использования применяются для выражения требуемого поведения разрабатываемой системы, без описания реализации этого поведения. Они позволяют разработчикам, конечным пользователям и экспертам в предметной области достичь взаимопонимания, а кроме того, помогают удостовериться в правильности архитектурных решений и проверять систему по ходу ее разработки. В процессе создания системы варианты использования реализуются с помощью кооперации, элементы которых работают совместно для достижения целей каждого из них.

Хорошо структурированные варианты использования описывают только существенные аспекты поведения и не являются ни слишком обобщенными, ни чересчур подробными.

Введение

Правильно спроектированный дом – это нечто намного большее, чем ряд стен, подпирающих крышу, которая защищает жильцов от непогоды. Работая вместе с архитектором над проектом дома, вы наверняка будете учитывать предполагаемое использование ваших помещений. Если вы любите принимать гостей, нужно продумать

план гостиной, чтобы людям было удобно общаться. Думая о приготовлении пищи для семьи, следует проектировать кухню таким образом, чтобы все шкафы и бытовая техника были размещены удобным образом. Даже маршрут транспортировки продуктов из машины на кухню в значительной мере повлияет на расположение комнат. Если у вас большая семья, надо заранее позаботиться о ванных комнатах. Определение их оптимального количества и размещения позволит избежать «очередей» по утрам, когда все одновременно собираются в школу или на работу. Если в семье есть подростки, это особенно актуально, поскольку цена эмоциональных стрессов высока.

Размышления о том, как вы и ваша семья будете распоряжаться домом, – это пример анализа вариантов использования. Вы рассматриваете разные способы использования дома, которые в итоге обуславливают его архитектуру. Для многих семей варианты использования схожи: во всех домах едят, спят, растят детей и хранят воспоминания. Но в каждом случае выдвигаются и индивидуальные требования к жилищу. Потребности большой семьи, например, будут отличаться от запросов молодого человека, только что окончившего колледж. И эти различия окажут решающее влияние на то, как будет выглядеть готовый дом.

Важнейшая особенность разработки вариантов использования (вроде вышеописанного) состоит в том, что вы не специфицируете конкретный способ их реализации. Например, поведение банкомата можно описать посредством вариантов взаимодействия с ним пользователей, но вам необязательно при этом знать, что у него внутри. Варианты использования специфицируют внешнее поведение, ничего не говоря о том, как его достичь. Это очень важно, потому что позволяет вам как эксперту или конечному пользователю общаться с разработчиками, конструирующими систему в соответствии с вашими требованиями, не углубляясь в детали реализации. Подробности будут рассмотрены позже, а на данном этапе вы можете сосредоточиться на наиболее существенных проблемах.

В UML поведение моделируется посредством вариантов использования, специфицируемых независимо от реализации. *Вариант* использования – это описание множества последовательных действий (включая вариации), которые выполняются некоторым субъектом с целью получения результата, значимого для некоторого действующего лица. Это определение включает в себя несколько важных пунктов.

Взаимодействия обсуждаются в главе 16, требования – в главе 6.

На системном уровне вариант использования описывает набор последовательностей, каждая из которых представляет взаимодействие сущностей, находящихся вне системы (действующих лиц), с самой системой и ее ключевыми абстракциями. Такие взаимодействия в действительности являются функциями уровня системы, которыми вы пользуетесь для визуализации, специфицирования,

конструирования и документирования ее ожидаемого поведения на этапах сбора и анализа требований к системе в целом. Например, один из основных вариантов использования в работе банка – это обработка займов.

Вариант использования предполагает взаимодействие действующих лиц и системы или другого субъекта. *Действующее лицо* представляет собой логически связанное множество ролей, которые играют пользователи системы во время взаимодействия с ней. Действующими лицами могут быть как люди, так и автоматизированные системы. Например, при моделировании работы банка процесс обработки займов включает в себя, помимо всего прочего, взаимодействие клиента с сотрудником кредитного отдела.

Варианты использования могут иметь разновидности. В любой системе существуют варианты использования, которые либо являются специализированными версиями других, более общих, либо входят в состав других вариантов использования, либо расширяют их поведение. Вы можете выделить общее, повторно применяемое поведение из множества вариантов использования, организуя их в соответствии с этими тремя видами связей. Например, при моделировании работы банка базовый вариант использования, описывающий обработку займов, подразумевает несколько разновидностей – от оформления крупной закладной до выдачи маленькой деловой ссуды. Притом все эти варианты использования имеют нечто общее в особенностях поведения (например, оценку платежеспособности клиента).

Всякий вариант использования должен выполнять некоторый объем работы. С точки зрения действующего лица он делает нечто представляющее определенную ценность: например, вычисляет результат, создает новый объект или изменяет состояние другого объекта. В примере с работой банка процесс обработки заявки на ссуду приводит к подписанию расходного документа и материализуется в виде некоторой суммы денег, вручаемой клиенту.

Подсистемы описываются в главе 32, классы – в главах 4 и 9, интерфейсы – в главе 11.

Вы можете применять варианты использования ко всей системе или к ее частям, в том числе к подсистемам и даже к индивидуальным классам и интерфейсам. В каждом случае варианты использования не только представляют желаемое поведение этих элементов, но также могут служить основой сценариев тестирования на различных этапах разработки. Далее, в применении к подсистемам это отличный источник регрессионных тестов, а в применении к системе в целом – комплексных и системных тестов. Варианты использования и действующие лица в UML изображаются, как показано на рис. 17.1. Эта нотация позволяет визуализировать определенный вариант использования в контексте других и отдельно от его реализации.

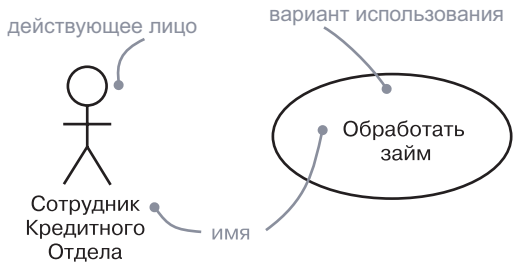


Рис. 17.1. Действующие лица и варианты использования

Базовые понятия

Нотация вариантов использования подобна нотации кооперации (см. главу 28).

Вариант использования (use case) – это описание множества последовательных действий, включая их варианты, выполняемых системой с целью получения значимого результата для действующего лица. Изображается в виде эллипса.

Субъект

Субъект – это класс, описанный набором вариантов использования. Обычно речь идет о системе или подсистеме. Варианты использования представляют аспекты поведения класса. Действующие лица же представляют аспекты других классов, взаимодействующих с субъектом. Взятые вместе, варианты использования описывают полное поведение субъекта.

Имена

Имя варианта использования должно быть уникальным в пределах включающего его пакета (см. главу 12).

Каждый вариант использования должен иметь имя, отличающее его от прочих. Имя варианта использования представляет собой текстовую строку, которая, взятая сама по себе, называется *простым именем*. К *квалифицированному имени* добавляется префикс – имя пакета, в котором находится вариант использования. Обычно при изображении варианта использования указывается только его имя, как показано на рис. 17.2.

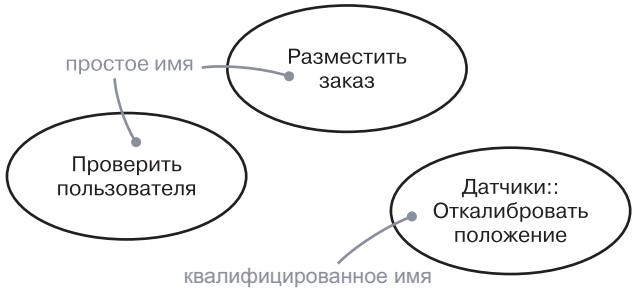


Рис. 17.2. Простое и квалифицированное имена

На заметку. В имени варианта использования могут присутствовать любые буквы латинского алфавита и цифры в неограниченном количестве, а также большинство знаков препинания (за исключением таких как двоеточие, которое используется для разделения имен варианта использования и включающего его пакета). Имя может занимать несколько строк. На практике для именования вариантов использования используются короткие глагольные конструкции в действительном залоге, обозначающие некоторое поведение из словаря моделируемой системы.

Варианты использования и действующие лица

Действующее лицо представляет собой связанное множество ролей, которые исполняют пользователи вариантов использования во время взаимодействия с ними. Обычно действующее лицо представляет ту роль, которую в данной системе играет человек, аппаратное устройство или даже другая система. Например, если вы работаете в банке, то можете играть роль `LoanOfficer` (Сотрудник кредитного отдела). Если в этом банке открыт ваш счет, то вы выступаете в качестве `Customer` (Клиент). Таким образом, экземпляр действующего лица представляет собой конкретную личность, определенным образом взаимодействующую с системой. Хотя вы и используете действующие лица в своих моделях, они не являются частью системы, так как существуют вне ее.

В работающей системе действующие лица не обязаны присутствовать как отдельные сущности. Один объект может играть роль множества действующих лиц. Например, один и тот же человек (`Person`) может быть и сотрудником кредитного отдела (`LoanOfficer`), и клиентом (`Customer`).

Обобщения рассматриваются в главах 5 и 10.

Как показано на рис. 17.3, действующие лица изображаются в виде человеческих фигурок. Вы можете вводить общие типы действующих лиц, такие как `Customer`, и специализировать их (например, создать разновидность `CommercialCustomer` – коммерческий клиент), определив связи обобщения.

Ассоциации обсуждаются в главах 5 и 10, сообщения – в главе 16.

На заметку. Можно использовать механизм расширения UML, приписав действующему лицу стереотип, чтобы создать другую пиктограмму, больше подходящую для достижения ваших целей.

Стереотипы обсуждаются в главе 6.

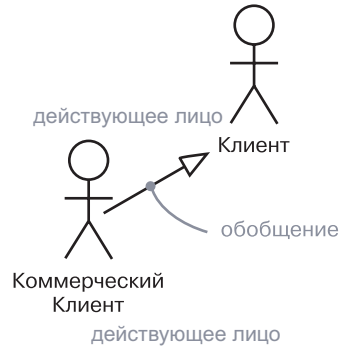


Рис. 17.3. Действующие лица

Действующие лица можно связывать с вариантами использования только при помощи ассоциаций. Ассоциация между действующим лицом и вариантом использования показывает, что они общаются друг с другом, возможно, посылая или принимая сообщения.

Варианты использования и поток событий

Вариант использования описывает, *что* делает система (или подсистема, или класс, или интерфейс), но не указывает, *как* она это делает. В процессе моделирования всегда важно разделять внешнее и внутреннее представления.

Можно специфицировать поведение варианта использования, описав поток событий в текстовой форме, понятной постороннему читателю. В описании должны присутствовать указание на то, как и когда вариант использования начинается и заканчивается, когда он взаимодействует с действующими лицами, какими объектами они обмениваются, а также упоминание основного и альтернативного потоков поведения.

Например, в контексте системы банкомата можно описать вариант использования `ValidateUser` (Проверка пользователя):

- *основной поток событий:* вариант использования начинается, когда система запрашивает у клиента его персональный идентификационный номер (PIN). Клиент (`Customer`) вводит его с клавиатуры. Завершается ввод нажатием клавиши Enter. Затем система проверяет введенный PIN и, если он правилен, подтверждает ввод. Этим вариант использования заканчивается;
- *исключительный поток событий:* клиент может отменить транзакцию в любой момент, нажав клавишу Cancel. Это действие запускает вариант использования заново. Никаких изменений со счетом клиента не производится;

- *исключительный поток событий:* клиент может в любой момент до нажатия клавиши Enter стереть свой PIN и ввести новый;
- *исключительный поток событий:* если клиент вводит неправильный PIN, вариант использования перезапускается. Если это происходит три раза подряд, система отменяет всю транзакцию и не позволяет данному клиенту работать с банкоматом в течение 60 с.

На заметку. Поток событий в варианте использования можно описать различными способами, в том числе в виде неформального структурированного текста (как в предыдущем примере), формального структурированного текста (с пред- и постусловиями), в виде конечного автомата (особенно для реактивных систем) либо с помощью псевдокода.

Варианты использования и сценарии

Диаграммы взаимодействия, включая диаграммы последовательности и коммуникации, рассматриваются в главе 19.

Обычно в начале работы вы будете описывать поток событий варианта использования в текстовой форме. Однако по мере уточнения требований к системе вам при изображении потоков понадобятся диаграммы взаимодействий, позволяющие представить их графически. Как правило, вы будете применять диаграмму последовательности для описания основного потока варианта использования, а ее вариации – для исключительных потоков.

Желательно отделять основной поток от альтернативных, поскольку вариант использования описывает не одну, а множество последовательностей, и выразить все детали интересующего вас варианта использования в виде одной последовательности невозможно. Например, в системе управления человеческими ресурсами присутствует вариант использования `Hire employee` (Нанять работника). Существует множество разновидностей этой основной бизнес-функции. Вы можете переманить работника из другой компании (наиболее общий сценарий), перевести сотрудника из одного подразделения в другое (что часто случается в транснациональных компаниях) или нанять иностранца (особый случай, регулируемый специальными правилами). Каждый из этих вариантов описывается своей последовательностью.

Экземпляры обсуждаются в главе 13.

`Hire employee` описывает набор последовательностей, каждая из которых представляет один поток из всех возможных вариаций. Такая последовательность называется сценарием. *Сценарий* (scenario) – это конкретная последовательность действий, иллюстрирующих поведение. Сценарии по отношению к вариантам использования – то же самое, что экземпляры по отношению к классам, поскольку сценарий – это в основном один экземпляр варианта использования.

На заметку. Существует фактор количественного роста от вариантов использования к сценариям. Система относительно небольшой сложности может включать несколько дюжин вариантов использования, определяющих ее поведение, и каждый вариант использования может включать до нескольких дюжин сценариев. У каждого варианта использования вы найдете главные сценарии (которые определяют существенные последовательности) и второстепенные (определяющие альтернативные последовательности).

Варианты использования и кооперации

Кооперации описаны в главе 28.

Вариант использования определяет поведение системы (или подсистемы, класса или интерфейса), которую вы разрабатываете, не указывая того, как это поведение реализовано. Разделение спецификации поведения и его реализации важно, потому что анализ системы, специфицирующий поведение, должен по возможности быть независимым от решений реализации, которые специфицируют, *как* именно оно обеспечивается. В конечном счете, однако, вы должны реализовать варианты использования путем создания сообществ классов и других элементов, работающих вместе для реализации поведения, описанного вариантом использования. Такое сообщество элементов, обладающее как статической, так и динамической структурой, моделируется в UML как кооперация.

Реализации обсуждаются в главах 9 и 10.

На рис. 17.4 показано, что реализацию варианта использования можно специфицировать явно через кооперацию. Поскольку большую часть времени данный вариант использования реализуется только одной кооперацией, нет необходимости моделировать эту связь явно.

На заметку. Хотя вы можете и не визуализировать такую эту связь явно, инструментальные средства, вероятно, ее поддерживают.

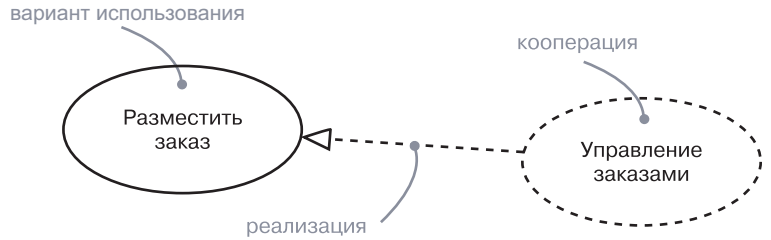


Рис. 17.4. Варианты • использования и кооперации

Архитектура обсуждается в главе 2.

На заметку. Нахождение минимального набора хорошо структурированных кооперации, удовлетворяющих потоку событий, специфицированному во всех вариантах использования системы, – основная задача системной архитектуры.

Организация вариантов использования

Пакеты обсуждаются в главе 12.

Для организации вариантов использования их группируют в пакеты так же, как классы. Кроме того, вы можете организовать варианты использования, определив между ними связи обобщения, включения и расширения. Эти связи применяются для того, чтобы выделить некоторое общее поведение (извлекая его из других вариантов использования), а также разновидности (помещая такое поведение в другие варианты использования, которые расширяют данный).

Обобщения обсуждаются в главах 5 и 10.

Обобщения между вариантами использования опять-таки подобны обобщениям между классами. Это означает, что дочерний вариант использования наследует поведение и суть родительского варианта использования; потомок может добавить или переопределить поведение родителя, а кроме того, быть подставленным вместо него в любом месте, где тот появляется (как родительский, так и дочерний вариант использования могут иметь конкретные экземпляры). Например, в банковской системе может существовать вариант использования `Validate User` (Проверка пользователя), который отвечает за идентификацию клиента. У него могут быть два специализированных дочерних варианта использования – `Check password` (Проверка пароля) и `Retinal scan` (Сканирование сетчатки), каждый из которых ведет себя так же, как `Validate User`, и может быть применен в любом месте, где появляется последний. При этом оба потомка добавляют свое собственное поведение: первый проверяет текстовый пароль, а второй – уникальный рисунок сетчатки глаза пользователя. Как показано на рис. 17.5, обобщение между вариантами использования изображается сплошной линией с большой треугольной стрелкой, то есть так же, как обобщение между классами.

Связь включения между вариантами использования означает, что базовый вариант использования в определенном месте явно включает в себя поведение некоторого другого . Включенный вариант использования не существует отдельно: он является экземпляром только внутри базового, который его содержит. Можно считать, что базовый вариант использования заимствует поведение включаемого.

Благодаря наличию связей включения удается избежать многократного описания одного и того же потока событий, поскольку общее поведение можно представить в виде отдельного варианта использования, включаемого в базовые. Связь включения являет

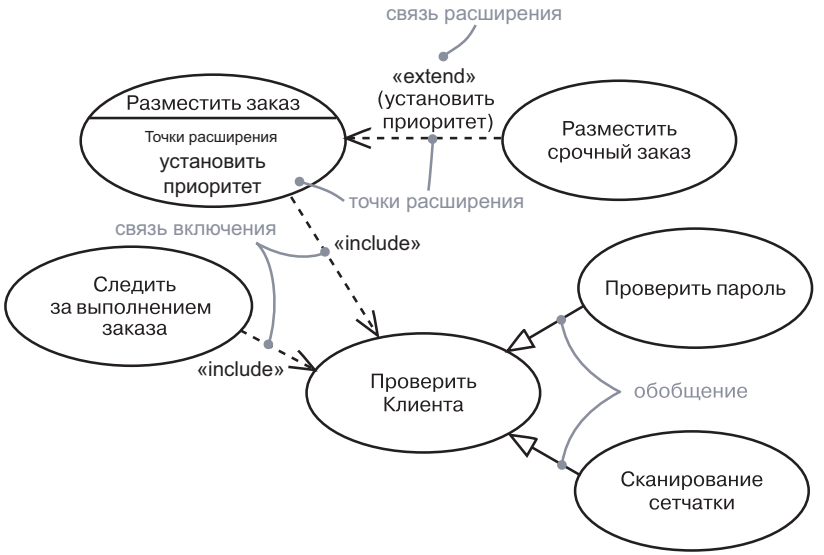


Рис. 17.5. Обобщение, включение и расширение

собой пример делегирования, когда ряд обязанностей системы описывается в одном месте (во включаемом варианте использования), а остальные варианты использования при необходимости вводят эти обязанности в свой набор.

Связи зависимости обсуждаются в главах 5 и 10, стереотипы – в главе 6.

Связь включения изображается как зависимость со стереотипом `include`. Чтобы показать то место в потоке событий базового варианта использования, куда включается поведение другого варианта использования, просто напишите `include` с последующим именем включаемого варианта использования, как это сделано, например, в потоке `Track order` (Отследить заказ):

```
Track order:
  Получить и проверить номер заказа;
  include 'Validate user';
  для каждой части заказа
    запросить состояние;
  сообщить общее итоговое состояние пользователю.
```

На заметку. В UML не предусмотрена готовая нотация для выражения сценариев вариантов использования. Используемый здесь синтаксис заимствован из структурированного естественного языка. Некоторые разработчики полагают, что неформальная нотация даже лучше, поскольку варианты использования не являются жесткими спецификациями, предназначенными для автоматической генерации кода. Другие же предпочитают формальную нотацию.

Связь расширения между вариантами использования означает, что базовый неявно включает поведение некоторого другого в косвенно указанном месте. Базовый вариант использования способен существовать отдельно, но при некоторых условиях его поведение может быть расширено поведением другого варианта использования. Базовый вариант использования можно расширить только вызовом из определенной точки, – так называемой *точки расширения* (extension point). Чтобы наглядно представить ситуацию, вообразите, что расширяющий вариант использования «вталкивает» поведение в базовый.

Связь расширения используется при моделировании тех частей вариантов использования, которые пользователь видит как необязательные. Таким образом обязательное поведение отделяется от необязательного. Также вы вправе применять связь расширения, чтобы выделить настраиваемые части реализуемой системы; следовательно, система может существовать как с различными расширениями, так и без них.

Связь зависимости обсуждается в главах 5 и 10, стереотипы и дополнительные секции – в главе 6.

Связь расширения изображается как зависимость со стереотипом `extend`. В дополнительной секции можно перечислить точки расширения базового варианта использования. Эти точки расширения – простые метки, которые могут появляться в потоке базового варианта использования. В качестве примера рассмотрим поток `Place order` (Разместить заказ):

```
Place order:
  include Validate user ;
  собрать компоненты пользовательского заказа;
  установить приоритет: точка расширения;
  подтвердить заказ для обработки.
```

В этом примере установить приоритет – точка расширения. Вариант использования может иметь несколько точек расширения, каждая из которых может встречаться не один раз, с обязательным указанием их имени. В обычных условиях этот базовый вариант использования будет исполняться без учета приоритетности заказа. С другой стороны, если это экземпляр приоритетного заказа, то поток идет, как описано выше, но в точке расширения `установить приоритет` вставлен расширяющий вариант использования `Place rush order` (Разместить срочный заказ), после которого поток продолжает исполнение. Если есть множество точек расширения, то в каждую просто вставляется расширяющий вариант использования.

На заметку. Организация вариантов использования, предусматривающая извлечение общего поведения (через связь включения) и разделение вариаций (через связь расширения) – важная составляющая создания простого, сбалансированного и понятного набора вариантов использования системы.

Прочие средства

Атрибуты и операции обсуждаются в главе 4, автоматы – в главе 22.

Варианты использования – это классификаторы, поэтому они могут иметь атрибуты и операции, которые можно изображать так же, как в классах. Можно представить атрибуты в виде объектов, находящихся внутри варианта использования и необходимых для описания его внешнего поведения, а операции – как действия системы, которые нужны для описания потока событий. Эти объекты и операции могут использоваться в диаграммах взаимодействия для спецификации поведения варианта использования.

Будучи классификаторами, варианты использования допускают присоединение к ним автоматов. Вы можете применять автоматы в качестве дополнительного средства описания поведения, представленного вариантом использования.

Типичные приемы моделирования

Моделирование поведения элемента

Системы и подсистемы обсуждаются в главе 32, классы – в главах 4 и 9.

Наиболее общий случай применения вариантов использования – это моделирование поведения элемента (системы в целом, подсистемы или класса). В такой ситуации необходимо сосредоточить внимание на том, что элемент делает, а не на том, как он это делает.

Подобное применение вариантов использования к элементам важно по трем причинам. Во-первых, вы позволяете экспертам в предметной области специфицировать внешнее представление элемента так, чтобы разработчикам было этого достаточно для конструирования его внутреннего представления. Обеспечивается возможность общения экспертов в предметной области с конечными пользователями и разработчиками. Во-вторых, разработчики получают возможность выработать определенный подход к элементу и понять его. Система, подсистема или класс подчас сложны и включают множество операций и других частей. Специфицируя вариант использования элемента, вы помогаете его пользователям постичь его самым непосредственным образом, в соответствии с тем, как они его собираются его применять. В противном случае пользователи вынуждены самостоятельно исследовать способы применения элемента. В-третьих, варианты использования служат базой для тестирования каждого элемента в процессе его разработки. Создавая тесты на их основе и применяя их регулярно, вы можете постоянно проверять работоспособность реализации. Но варианты использования не только участвуют в разработке регрессионных тестов: всякий раз, добавляя новый вариант использования элемента, вы тем самым вынуждаете пересматривать реализацию, чтобы

гарантировать достаточную гибкость элемента и его устойчивость к изменениям. Если окажется, что это не так, придется вносить соответствующие изменения в архитектуру.

Чтобы смоделировать п оведение элемента, необходимо:

- Идентифицировать действующие лица, взаимодействующие с элементом. Кандидаты на включение в эту группу – те, кто нуждается в определенном поведении элемента для выполнения своих собственных задач, либо те, кто прямо или косвенно задействован в функционировании элемента.
- Организовать действующие лица, определив общие и более специализированные роли.
- Рассмотреть основные пути взаимодействия каждого действующего лица с элементом, а также сами взаимодействия, которые изменяют состояние элемента или его окружения либо обеспечивают реакцию на некое событие.
- Рассмотреть исключительные пути взаимодействия каждого действующего лица с элементом.
- Организовать поведение, выявленное на этапах 3 и 4, в виде вариантов использования, применяя связи включения и расширения, чтобы выделить общее поведение и отделить исключительное.

Например, система розничной торговли должна взаимодействовать с заказчиками, которые размещают заказы и отслеживают их исполнение. Сама система, в свою очередь, отгружает заказанные товары и выставляет счета клиентам к оплате. Как видно из рис. 17.6,

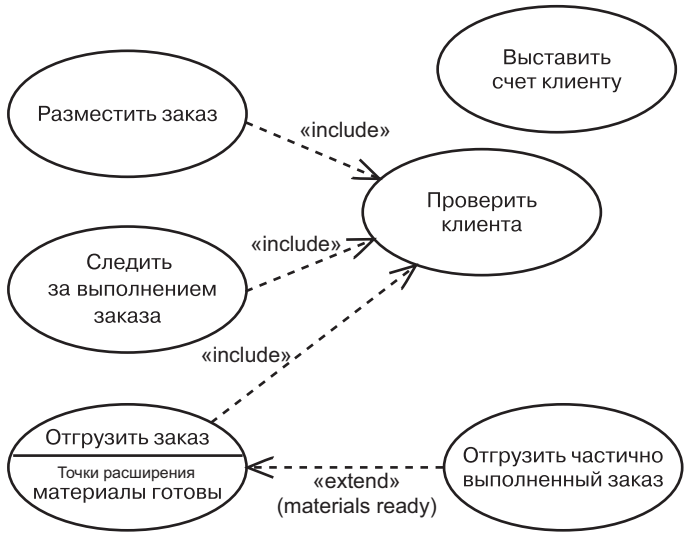


Рис. 17.6. Моделирование поведения элемента

поведение такой системы можно смоделировать, объявив несколько вариантов использования: `Place order` (Разместить заказ), `Track order` (Отследить заказ), `Ship order` (Отгрузить заказ) и `Bill customer` (Выставить счет). Можно выделить общее поведение `Validate customer` (Проверить клиента) и варианты типа `Ship partial order` (Отгрузить частично выполненный заказ). Для каждого из этих вариантов использования следует предусмотреть спецификацию поведения, выраженную текстом, автоматом или взаимодействием.

Пакеты обсуждаются в главе 12.

По мере развития модели вы увидите целесообразность объединения вариантов использования в концептуально и семантически близкие группы. Такие группы в UML можно моделировать в виде пакетов.

Советы и подсказки

При моделировании вариантов использования в UML каждый из них должен представлять некоторое отдельное и идентифицируемое поведение системы или ее части. Хорошо структурированный вариант использования обладает следующими свойствами:

- именует простое, идентифицируемое и атомарное (в той степени, в какой это целесообразно) поведение системы или ее части;
- выделяет общее поведение, извлекая его из других вариантов использования;
- выделяет вариации, помещая некоторое поведение в другие варианты использования, расширяющие его;
- описывает поток событий так, чтобы сделать его понятным постороннему читателю;
- описывается минимальным набором сценариев, специфицирующих его основную семантику и семантику вариаций.

Когда вы изображаете вариант использования в UML:

- показывайте только те варианты использования, которые важны для понимания поведения системы или ее части в данном контексте;
- показывайте только действующие лица, связанные с этими вариантами использования.

Глава 18. Диаграммы вариантов использования

В этой главе:

- Моделирование контекста системы
- Моделирование требований к системе
- Прямое и обратное проектирование

Диаграммы деятельности обсуждаются в главе 20, диаграммы состояний – в главе 25, диаграммы последовательности и коммуникации – в главе 19.

Диаграммы вариантов использования – это один из видов диаграмм UML, предназначенных для моделирования динамических аспектов систем. (Остальные четыре вида с аналогичным назначением – это диаграммы деятельности, состояний, последовательности и коммуникации). Диаграммы вариантов использования – основной вид диаграмм при моделировании поведения системы, подсистемы или класса. Каждая из них показывает набор вариантов использования и действующих лиц в их взаимодействии.

Диаграммы вариантов использования применяются для моделирования представления системы с точки зрения вариантов использования. Большей частью это подразумевает моделирование контекста системы, подсистемы или класса либо моделирование требований к этим элементам.

Диаграммы вариантов использования важны для визуализации, специфицирования и документирования поведения элемента. Они обеспечивают доступность и понятность систем, подсистем и классов за счет внешнего представления того, как эти элементы могут быть использованы в контексте. Кроме того, такие диаграммы важны для тестирования работающих систем посредством прямого проектирования и для обеспечения их понимания посредством обратного проектирования.

Введение

Предположим, вам вручили техническое устройство, которое вы прежде в глаза не видели. С одной стороны небольшого корпуса расположены какие-то кнопки и жидкокристаллическая панель.

Однако устройство не имеет никакого описания, а вы даже не догадываетесь о том, как его использовать. Можно в случайном порядке нажимать кнопки и смотреть, что произойдет, но так вы потратите уйму времени на метод проб и ошибок, а вот достигнете ли результата – нельзя сказать наверняка.

То же самое с программными системами. Если вы пользователь, приступающий к работе с новым приложением, вам нужны четкие инструкции. Если приложение следует стандартным соглашениям, принятым в операционной системе, к которой вы привыкли, вам будет немного проще освоить интерфейс, и все же только на основании этого вы вряд ли поймете более тонкие и сложные нюансы поведения программы. Аналогичная ситуация: если вы разработчик, вам могут передать приложение, которое делал ваш коллега, и попросить в нем разобраться. Выполнить эту задачу, пока вы не сформируете концептуальную модель его применения, будет практически невозможно.

В UML диаграммы вариантов использования применяются для того, чтобы визуализировать поведение системы, подсистемы или класса с тем, чтобы пользователь мог понять, как применять этот элемент, а разработчик – как реализовать его. Диаграмма вариантов использования, показанная на рис. 18.1, помогает смоделировать поведение того самого «загадочного» устройства с кнопками и дисплеем, в котором большинство людей без труда узнает сотовый телефон.

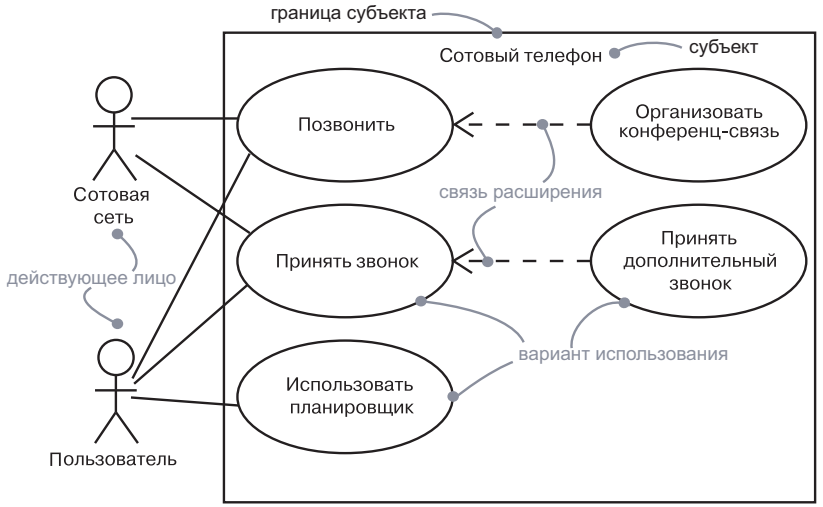


Рис. 18.1. Диаграмма вариантов использования

Базовые понятия

Диаграмма вариантов использования – это диаграмма, которая показывает набор вариантов использования и действующих лиц, а также их связи.

Общие свойства

Общие свойства диаграмм обсуждаются в главе 7.

Диаграмма вариантов использования, как и любая другая, обладает именем и графическим наполнением, представляющим собой проекцию модели. От всех других видов диаграмм отличается конкретным содержанием.

Содержание

Варианты использования и действующие лица обсуждаются в главе 17, связи – в главах 5 и 10, пакеты – в главе 12, экземпляры – в главе 13.

Обычно диаграммы вариантов использования содержат субъект, варианты использования, действующие лица, а также связи зависимости, обобщения и ассоциации. Как и все другие диаграммы, могут содержать примечания и ограничения.

Кроме того, диаграммы вариантов использования в ряде случаев включают пакеты, служащие для объединения элементов модели в большие группы. Иногда вам потребуется поместить на диаграмму экземпляры вариантов использования, особенно если вы хотите визуализировать конкретную систему.

Нотация

Субъект изображается в виде прямоугольника, содержащего набор эллипсов (вариантов использования). Имя субъекта указано внутри прямоугольника. Действующие лица представлены в виде фигурок рядом с прямоугольником. Имена действующих лиц располагаются под фигурками. Последние соединяются линиями с эллипсами вариантов использования, с которыми они взаимодействуют. Связи между вариантами использования, такие как расширение и включение, рисуются внутри прямоугольника.

Стандартное использование

Представления вариантов использования обсуждаются в главе 2.

Диаграммы вариантов использования нужны для того, чтобы моделировать представление субъекта (например, системы) с точки зрения вариантов использования. Обычно оно моделирует внешнее поведение субъекта, то есть видимые извне услуги, которые субъект предоставляет в контексте его окружения.

Моделируя представление субъекта с точки зрения вариантов использования, вы обычно применяете диаграммы вариантов использования одним из двух способов:

1. *Для моделирования контекста субъекта*, которое подразумевает очерчивание границ вокруг всей системы и определение действующих лиц, которые находятся вне нее и с ней взаимодействуют. Здесь диаграммы вариантов использования требуются для спецификации действующих лиц и значения их ролей.
2. *Для моделирования требований к субъекту*, которое подразумевает спецификацию того, что он должен делать (с точки зрения внешней по отношению к субъекту), независимо от того, как он должен это делать. Здесь диаграммы вариантов использования применяются для специфицирования требуемого поведения субъекта. В метафорическом смысле диаграмма позволяет вам представить субъекта как «черный ящик»: вы можете видеть, что происходит вне его и как он реагирует на внешние воздействия, но не видите, как он работает внутри.

Требования обсуждаются в главах 4 и 6.

Типичные приемы моделирования

Моделирование контекста системы

Какую бы систему мы ни обсуждали, легко заметить, что некоторые элементы находятся внутри нее, а некоторые – снаружи. Например, в системе проверки кредитных карт вы обнаружите такие сущности, как счета, транзакции и агенты предотвращения мошенничеств, а за ее пределами – такие сущности, как владельцы кредитных карт и системы розничной торговли, предусматривающие оплату по картам. Те элементы, которые «живут» внутри системы, отвечают за поведение, которое от нее ожидается. Все, что находится вне системы и взаимодействует с ней, составляет ее *контекст* (context). Контекст определяет среду, в которой живет система.

Системы обсуждаются в главе 32.

В UML вы можете моделировать контекст системы с помощью диаграммы вариантов использования, изображающей действующие лица, окружающие систему. Решение о том, что именно включить в диаграмму в качестве действующих лиц, важно постольку, поскольку вы таким образом специфицируете класс сущностей, взаимодействующих с системой. Решение о том, что не включать в качестве действующих лиц, в равной степени (если не более) важно, ибо оно ограничивает окружающую среду системы с тем, чтобы включить лишь действующие лица, действительно необходимые для ее жизни.

Чтобы смоделировать контекст системы, необходимо:

- Идентифицировать границы системы, приняв решение о том, какое поведение является ее частью, а какое осуществляют внешние сущности. Этим определяется субъект.
- Идентифицировать действующие лица, окружающие систему, рассмотрев при этом, какие группы требуют помощи со стороны системы в выполнении своих задач, какие необходимы для обеспечения функционирования системы, какие взаимодействуют с внешним оборудованием или другими программными системами и какие осуществляют вторичные функции по администрированию и поддержке.
- Организовать схожие действующие лица в иерархии обобщения–специализации.
- Там, где это требуется для лучшего понимания модели, представить стереотип для каждого из действующих лиц.

Наполните диаграмму вариантов использования этими действующими лицами и опишите пути взаимодействия каждого из них с вариантами использования системы.

В примере на рис. 18.2 показан контекст системы проверки кредитных карт (`Credit Card Validation System`), с описанием действующих лиц, окружающих ее. Вы видите клиентов (`Customers`), которые подразделяются на две группы: `Individual customer` (Частный клиент) и `Corporate customer` (Корпоративный клиент). Эти действующие лица представляют роли, которые играют люди при взаимодействии

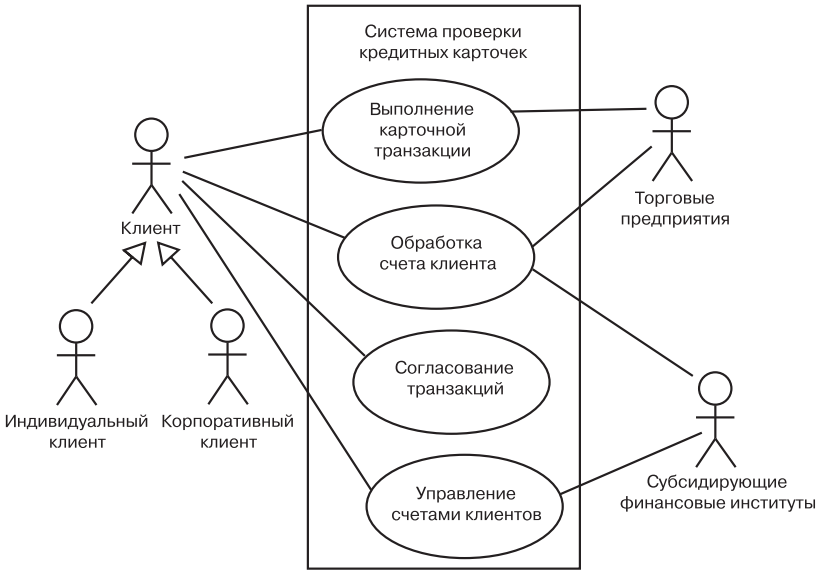


Рис. 18.2. Моделирование контекста системы

с системой. В данном контексте показаны также действующие лица, представляющие другие учреждения, – такие как `Retail Institution` (Предприятие розничной торговли), с которым `Customer` взаимодействует посредством карточной транзакции, приобретая товар или услугу, и `Sponsoring financial Institution` (Институт финансового спонсирования) – депозитный центр для карточных счетов. В действительности последние два, скорее всего, будут представлены программными системами.

Подсистемы обсуждаются в главе 32.

Те же приемы применимы для моделирования контекста подсистем. То, что является системой на одном уровне абстракции, часто предстает составляющей частью более крупной системы на другом, более высоком. Поэтому моделирование контекста подсистем полезно, когда вы разрабатываете систему с «вложениями».

Моделирование требований к системе

Требование определяет функцию, свойство или поведение системы. Формулируя требования к системе, вы составляете контракт между ней и теми сущностями, которые находятся вне ее. Этот контракт декларирует то, что система должна делать. Большей частью вас не заботит то, как она это будет делать – важно, чтобы она отвечала заявленным требованиям. Хорошая система выполняет требования точно, предсказуемо и надежно. Приступая к построению системы, важно начать с соглашений о том, что она должна делать, хотя, скорее всего, в процессе ее реализации вы будете не раз уточнять эти требования. Аналогичным образом, когда вы вводите систему в эксплуатацию, знание ее поведения весьма существенно для ее правильного использования.

Примечания можно применять для формулировки требований, как показано в главе 6.

Требования могут быть выражены в различных формах – от неструктурированного текста до выражений формального языка. Большинство функциональных требований к системе, если не все они, могут быть выражены в виде вариантов использования, а потому для управления этими требованиями важны диаграммы вариантов использования UML.

Чтобы смоделировать требования к системе, необходимо:

- Установить контекст системы, идентифицируя действующие лица, окружающие ее.
- Рассмотреть поведение системы, которого от нее ожидает или требует каждое действующее лицо.
- Обобщить это поведение в виде вариантов использования.
- Выделить общее поведение в новые варианты использования, на которые ссылаются другие варианты; выделить разновидности поведения в новые варианты использования, расширяющие основные потоки.

- Смоделировать эти варианты использования, действующие лица и их связи на диаграмме вариантов использования.
- Снабдить варианты использования примечаниями или ограничениями, которые задают нефункциональные требования; возможно, некоторые из них вы присоедините ко всей системе.

Моделирование динамики для распределения нагрузки и реконфигурации сети обсуждается в главе 24.

Рис. 18.3 дополняет диаграмму вариантов использования, показанную на предыдущем рисунке. Здесь скрыты связи между действующими лицами и вариантами использования, но зато вводятся дополнительные варианты использования, которые не видимы обычному пользователю, хотя и представляют важные аспекты поведения системы. Диаграмма полезна тем, что может послужить для конечных пользователей, экспертов в предметной области и разработчиков общей отправной точкой в процессе визуализации, специфицирования, конструирования и документирования их совместных решений, касающихся функциональных требований к системе. Например, `Detect card fraud` (Обнаружение мошенничеств) – это поведение, которое важно как для предприятий розничной торговли, так и для институтов финансового спонсирования. `Report on account status` (Отчет о состоянии счета) – еще одно поведение, требуемое от системы различными учреждениями в ее контексте.

Рис. 18.3. Моделирование требований к системе

Требование, которое моделируется вариантом использования `Manage network outage` (Управление сетевыми потоками), несколько отлично от всех остальных, поскольку описывает вторичное

поведение системы, необходимое для обеспечения ее надежной непрерывной работы.

Определив структуру варианта использования, вы должны описать поведение каждого из них. Обычно нужно составить одну или несколько диаграмм последовательности для каждого основного сценария, затем – диаграммы последовательности для вариаций сценариев и, наконец, хотя бы одну диаграмму последовательности, иллюстрирующую каждый из возможных типов ошибок или исключений. Обработка ошибок – часть варианта использования, поэтому она должна быть запланирована наряду с нормальным поведением.

Подсистемы обсуждаются в главе 32.

Те же приемы применяются при моделировании подсистем.

Прямое и обратное проектирование

Диаграммы обсуждаются в главе 7, варианты использования – в главе 14.

Большинство других диаграмм UML, включая диаграммы классов, компонентов, состояний, – явные кандидаты на прямое и обратное проектирование, потому что каждая из них имеет некий аналог в исполняемой системе. Диаграммы вариантов использования в этом смысле стоят слегка особняком, потому что они скорее отражают, чем специфицируют реализацию системы, подсистемы или класса. Варианты использования описывают то, как ведет себя элемент, а не то, как он реализован, поэтому они не могут быть подвергнуты прямому и обратному проектированию.

Прямое проектирование (forward engineering) – это процесс трансформации модели в код путем ее отображения на язык реализации. Диаграмма вариантов использования может быть подвергнута прямому проектированию с целью формирования тестов для того элемента, который она описывает. Каждый вариант использования на такой диаграмме специфицирует поток событий с его вариациями, и все эти потоки описывают ожидаемое поведение элемента – то есть нечто такое, что подлежит тестированию. Хорошо структурированный вариант использования даже специфицирует пред- и постусловия, которые могут применяться в целях определения начального состояния теста и критериев его успешности. Для каждого варианта использования из диаграммы вы можете создать *сценарий тестирования* (test case), который будет выполняться при выпуске каждой новой версии данного элемента, таким образом подтверждая, что он работает надлежащим образом, прежде чем другие элементы смогут опираться на него в своей работе.

Чтобы осуществить прямое проектирование диаграммы вариантов использования, необходимо:

- Идентифицировать объекты, которые взаимодействуют с системой. Попытайтесь идентифицировать различные роли, которые может играть каждый внешний объект.
- Создать действующее лицо, представляющее каждую из отдельных взаимодействующих ролей.
- Для каждого варианта использования на диаграмме идентифицировать основной и исключительный потоки событий.
- В зависимости от того, насколько тщательно вы собираетесь проводить тестирование, сгенерировать соответствующий сценарий для каждого потока, используя предусловие потока как начальное состояние теста и постусловие – как критерий успешности.
- При необходимости сгенерировать тестовую инфраструктуру, которая будет представлять каждое из действующих лиц, взаимодействующих с вариантом использования. Действующие лица, вводящие информацию для элемента либо производящие над ним какие-то действия, могут быть симулированы, или вместо них понадобится подставить некий эквивалент из реального мира.
- Использовать инструментальные средства для запуска каждого из этих тестов всякий раз, когда вы реализуете элемент, к которому относится диаграмма вариантов использования.

Обратное проектирование (reverse engineering) – процесс трансформации кода в модель посредством его отображения из определенного языка реализации. Автоматически выполнить обратное проектирование диаграммы вариантов использования на данном этапе не представляется возможным, потому что на пути от спецификации поведения элемента к его реализации происходит потеря информации. Однако в ваших силах изучить существующую систему и распознать ее подразумеваемое поведение самостоятельно, с тем чтобы на этой основе оформить диаграмму вариантов использования. В любом случае неплохо это делать каждый раз, когда вы имеете дело с недокументированным программным обеспечением. Диаграммы вариантов использования UML просто предоставляют вам стандарт и выразительный язык, чтобы можно было зафиксировать то, что вы при этом обнаружите.

Чтобы осуществить обратное проектирование диаграммы вариантов использования, необходимо:

- Идентифицировать каждое действующее лицо, взаимодействующее с системой.
- Исследовать манеру взаимодействия каждого из них с системой, изменение ее состояния или состояния среды, реакцию на события.

- Проследить поток событий в исполняемой системе относительно каждого действующего лица, начав с основных потоков и во вторую очередь изучив альтернативные.
- Объединить в группы взаимосвязанные потоки, определяя соответствующие варианты использования. Рассмотреть варианты моделирования с применением связей расширения и моделирование похожих потоков с применением связей включения.
- Отобразить эти варианты использования и действующие лица на диаграмме вариантов использования и установить их связи.

Советы и подсказки

Когда вы создаете диаграммы вариантов использования в UML, помните, что каждая такая диаграмма – это просто графический срез статического представления вариантов использования системы. Отсюда следует, что нет необходимости в одной-единственной диаграмме, которая охватывала бы это представление целиком. Диаграммы вариантов использования, взятые в совокупности, формируют статическое представление вариантов использования системы; каждая из них в отдельности выражает только какой-то один аспект.

Хорошо структурированная диаграмма вариантов использования обладает следующими характеристиками:

- сконцентрирована на передаче одного аспекта представления системы с точки зрения вариантов использования;
- содержит только те варианты использования и действующие лица, которые важны для понимания данного аспекта;
- обеспечивает представление, соответствующее ее уровню абстракции: вы указываете только те дополнения (в частности, точки расширения), которые важны для понимания диаграммы;
- не настолько лаконична, чтобы читатель упустил из виду важную семантику.

Когда вы рисуете диаграмму вариантов использования:

- присваивайте ей имя, соответствующее ее назначению;
- расположите ее элементы так, чтобы пересекающихся линий было как можно меньше;
- организуйте элементы так, чтобы семантически близкие по поведению и ролям располагались рядом;
- используйте примечания и цветовое выделение для того, чтобы привлечь внимание читателя к важным особенностям;
- постарайтесь не показывать слишком много видов связей. Вообще, если вы задействуете сложные связи включения и расширения, вынесите их в отдельную диаграмму.

Глава 19. Диаграммы взаимодействия

В этой главе:

- Моделирование потоков управления по времени
- Моделирование потоков управления по организации
- Прямое и обратное проектирование

Диаграммы деятельности, диаграммы состояний и диаграммы вариантов использования – это три типа диаграмм, которые наряду с рассматриваемым в данной главе используются в UML для моделирования динамических аспектов систем. Они обсуждаются соответственно в главе 20, 25 и 18.

Диаграммы взаимодействия, в том числе диаграммы последовательности и коммуникации, используются в UML для моделирования динамических аспектов систем. В общих чертах диаграмма взаимодействия показывает взаимодействие ряда объектов, а также их связи и сообщения, которые могут передаваться между ними. Диаграммы последовательности и коммуникации – это диаграммы взаимодействия, первая из которых отражает временной порядок сообщений, а вторая – структурную организацию объектов, отправляющих и принимающих сообщения.

Большей частью под моделированием динамических аспектов системы применительно к диаграммам взаимодействия подразумевается моделирование конкретных или прототипных экземпляров классов, интерфейсов, компонентов и узлов наряду с передаваемыми между ними сообщениями – и все это в контексте сценария, иллюстрирующего некоторое поведение. Диаграммы взаимодействия могут существовать самостоятельно, чтобы визуализировать, специфицировать, конструировать и документировать динамику определенных сообществ объектов, или же использоваться для моделирования одного конкретного потока управления в пределах варианта использования.

Диаграммы взаимодействия важны не только для моделирования динамических аспектов систем, но и для конструирования исполняемых систем методом прямого и обратного проектирования.

Введение

Когда вы смотрите кино- или телефильм, то испытываете определенного рода иллюзию. Вам кажется, что на экране происходит непрерывное движение, как в жизни, а на самом деле вы видите серию статических картинок, воспроизводимую настолько быстро, что возникает подобный «обман зрения».

Во время съемки режиссеры и мультипликаторы применяют ту же технику последовательных срезов событий, создавая раскадровку основных сцен. Иными словами, они строят модель каждой сцены, достаточно подробную, чтобы передать свой замысел коллегам по съемочной группе. Фактически создание раскадровки – это основное содержание производственного процесса съемок, позволяющее команде визуализировать, специфицировать, сконструировать и задокументировать модель кинофильма в ее развитии – от начальной точки к выпуску в прокат.

Моделирование структурных аспектов системы обсуждается в частях II и III.

При разработке программных систем встает та же проблема: как смоделировать динамическое поведение? Представьте на мгновение, как можно визуализировать работающую систему. Если в вашем распоряжении есть интерактивный отладчик, соединенный с системой, то вы можете просмотреть участок памяти и увидеть, как его содержимое со временем изменяется. При необходимости можно даже наблюдать за отдельными интересующими вас объектами. Немного понаблюдав, вы увидите создание некоторых объектов, изменения значений их атрибутов, а потом уничтожение некоторых из них.

Возможности подобной визуализации динамических аспектов системы в достаточной мере ограничены, особенно если речь идет о распределенной системе с множеством параллельных потоков управления. С таким же успехом вы можете попытаться понять систему кровообращения человека, сосредоточив внимание на порции крови, которая перекачивается через один участок одной-единственной артерии в единицу времени. Более рациональный способ моделирования динамических аспектов системы заключается в том, чтобы строить «раскадровки сценариев» с включением взаимодействия определенных объектов и сообщений, которые могут передаваться между ними.

В UML такие «раскадровки» моделируются диаграммами взаимодействия. Последние, как показано на рис. 19.1, можно строить одним из двух способов: выделяя временной порядок сообщений и выделяя структурные связи между взаимодействующими объектами. В любом случае эти диаграммы семантически эквивалентны: можно конвертировать одну в другую без потери информации.

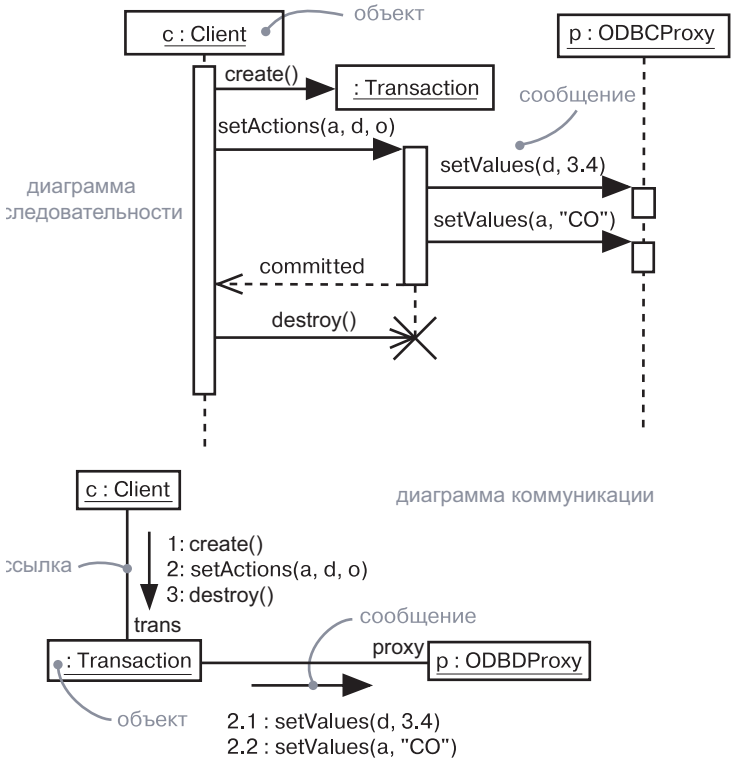


Рис. 19.1. Диаграммы взаимодействия

Базовые понятия

Диаграмма взаимодействия показывает взаимодействие, состоящее из набора объектов и их связей, включая передаваемые между ними сообщения.

Диаграмма последовательности – это диаграмма взаимодействия, которая подчеркивает временной порядок сообщений. Изображается как таблица, в которой представлены объекты, расположенные вдоль оси X, и сообщения, упорядоченные по ходу времени – вдоль оси Y.

Диаграмма коммуникации – это диаграмма взаимодействия, которая выделяет структурную организацию объектов, отправляющих и принимающих сообщения. Графически представляет собой набор дуг и вершин.

Общие свойства

Общие свойства диаграмм обсуждаются в главе 7.

Диаграмма взаимодействия, как и любая другая, характеризуется именем и содержимым, представляющим проекцию модели. От других типов диаграмм отличается конкретным наполнением.

Содержимое

Объекты обсуждаются в главе 13, ссылки – в главах 14 и 16, сообщения и взаимодействия – в главе 16, внутренняя структура класса – в главе 15, кооперации – в главе 28.

Диаграммы взаимодействия обычно содержат роли или объекты; коммуникации или ссылки; сообщения.

На заметку. Диаграммы взаимодействия, по существу, представляют проекцию элементов, участвующих во взаимодействии. Семантика контекста взаимодействия, объектов и ролей, ссылок и коннекторов, сообщений и последовательностей применима к диаграммам взаимодействия.

Диаграммы взаимодействия, как и все прочие, могут содержать примечания и ограничения.

Диаграммы последовательности

Диаграммы последовательности выделяют временной порядок сообщений. Как показано на рис. 19.2, формирование диаграммы последовательности начинается с размещения объектов или ролей, участвующих во взаимодействии, в верхней части диаграммы, по горизонтальной оси. Обычно объект или роль, инициирующие взаимодействие, размещаются слева, а зависимые объекты или роли – справа. Далее по вертикальной оси расставляются сообщения, которые отправляют и принимают эти объекты, – сверху вниз, в хронологическом порядке. Это создает у читателя представление о потоке управления во времени.

Диаграммы последовательности отличаются от диаграмм коммуникации двумя признаками.

Во-первых, это *линии жизни* (lifelines). *Линия жизни объекта* – вертикальная пунктирная линия, символизирующая существование объекта в течение некоторого периода времени. Большинство объектов, представленных на диаграмме взаимодействия, существуют в течение всего взаимодействия, поэтому все они выровнены по верхней границе диаграммы, а линии их жизни проведены от верха до низа.

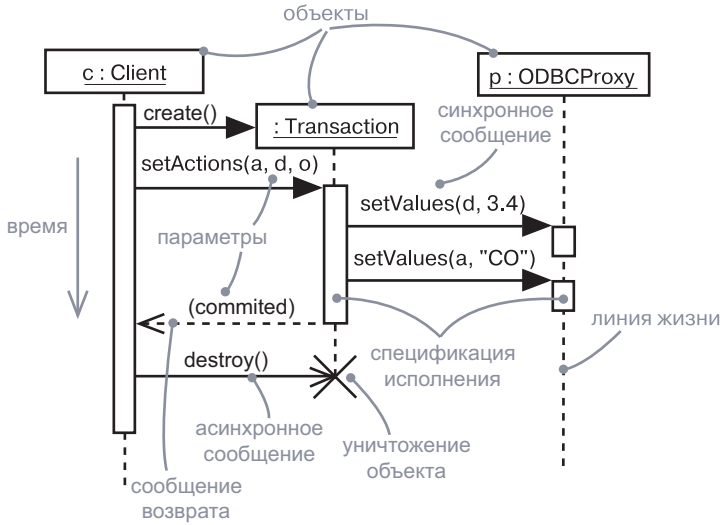


Рис. 19.2. Диаграмма последовательности

Объекты могут быть созданы в процессе взаимодействия. Их время жизни начинается с получения сообщения `create`, направленного к прямоугольнику объекта в начале жизненного пути. Равным образом в процессе взаимодействия объекты могут уничтожаться. Их линия жизни заканчивается при получении сообщения `destroy`, что графически отмечено большим символом X.

Если взаимодействие отражает историю конкретных объектов, то символ объекта с подчеркнутым именем размещается в начале линии жизни. Однако в основном вам придется показывать взаимодействия прототипов. При этом линии жизни не представляют конкретных объектов – они специфицируют прототипные роли, представляющие разные объекты в каждом экземпляре взаимодействия. В данном случае не нужно подчеркивать имена, поскольку они не символизируют конкретных объектов.

На заметку. Если объект изменяет значения своих атрибутов, свое состояние либо свои роли, то вы можете пометить такую модификацию, разместив пиктограмму состояния на линии жизни объекта в точке, где происходит изменение.

Во-вторых, это наличие фокуса управления. *Фокус управления* (focus of control) – высокий узкий прямоугольник, показывающий период времени, в течение которого объект выполняет действие – как непосредственно, так и с помощью зависимой процедуры. Верхняя

грань прямоугольника выровнена по началу действия, а нижняя – по его завершению и может быть отмечена сообщением возврата. Вы можете показать вложенность фокуса управления, вызванную рекурсией, вызовом собственной операции либо возвратом вызова из другого объекта, наложив другой фокус управления чуть правее родительского (таким образом можно изобразить сколько угодно уровней вложения). Если нужно особенно точно показать расположение фокуса управления, оттените часть прямоугольника, обозначающего период времени, в течение которого на самом деле работает метод объекта и управление не передается другому объекту. Правда, эта конструкция будет казаться довольно «утяжеленной».

Основное содержимое диаграммы последовательности – *сообщения*. Они изображаются стрелками, направленными от одной линии жизни к другой. Стрелка указывает на приемник сообщения. Если таковое асинхронно, то стрелка рисуется «уголком», а если синхронно (вызов), то закрашенным треугольником. Ответ на синхронное сообщение (возврат из вызова) показывается пунктирной стрелкой «уголком». Сообщение возврата может быть опущено, поскольку каждый вызов неявно подразумевает возврат, но иногда удобно таким образом продемонстрировать возвращаемое значение.

Упорядочение по времени вдоль единственной линии жизни весьма важно. Обычно точное расстояние не имеет значения; линии жизни показывают лишь относительные последовательности, поэтому не обеспечивают масштабного отображения времени. Кроме того, позиции сообщений на отдельных парах линий жизни, как правило, не влияют на хронологию передачи информации; сообщения могут поступать в любом порядке. Полные наборы сообщений на отдельных линиях жизни формируют частичное упорядочение. Серии сообщений, однако, устанавливают цепь причинных связей, поэтому любая точка на другой линии жизни в конце цепи должна всегда следовать за точкой начала цепи на исходной линии.

Структурированное управление на диаграммах последовательности

Последовательность сообщений прекрасно подходит для отображения отдельных линейных последовательностей, но часто возникает необходимость показать условные операторы и циклы. Иногда нужно отобразить параллельное выполнение множества последовательностей. Эти разновидности высокоуровневого управления могут быть показаны на диаграммах последовательности с помощью операторов структурированного управления.

Оператор управления (control operator) изображается на диаграмме последовательности в виде прямоугольной области. Он сопровождается тегом – текстовой меткой, заключенной в маленький пятиугольник, в верхнем левом углу. Она показывает, какой это оператор управления. Оператор применяется к линиям жизни, которые его пересекают. Эта часть называется телом оператора. Если линия жизни не затрагивается оператором, она может быть прервана в его начале (вверху) и продолжена в конце (внизу). Чаще всего применяются следующие виды управления:

- **необязательное выполнение** (тег `opt`). Тело этого управляющего оператора выполняется, если защитное условие истинно на его входе. Защитное условие – это булево выражение, которое указано в квадратных скобках в верхней части одной из линий жизни внутри тела и может ссылаться на атрибуты объекта;
- **условное выполнение** (тег `alt`). Тело управляющего оператора разделяется на несколько подобластей горизонтальными пунктирными линиями. Каждая подобласть представляет одну ветвь условия и снабжена своим защитным условием. Если защитное условие подобласти истинно, то она выполняется. Вместе с тем может выполниться не более одной подобласти. Если истинно несколько защитных условий, то выбор подобласти для выполнения не определен и может варьироваться от запуска к запуску. Если ни одно из условий не истинно, то управление продолжается за пределами оператора. Одна подобласть может иметь особое защитное условие `[else]`. Такая подобласть выполняется, если не истинны никакие другие защитные условия;
- **параллельное выполнение** (тег `par`). Тело управляющего оператора делится на несколько подобластей горизонтальными пунктирными линиями. Каждая подобласть представляет параллельный (конкурирующий) поток вычислений. При этом в большинстве случаев каждая подобласть включает разные линии жизни. Когда управление переходит к данному оператору, все его подобласти начинают выполняться параллельно. Выполнение сообщений в каждой из них последовательно, но относительный порядок сообщений из параллельных подобластей совершенно произволен. Эта конструкция не должна применяться, если разные вычисления взаимодействуют. Очень удобный оператор, так как существует очень много ситуаций из реального мира, в которых можно вычленить независимые параллельные потоки деятельности;

- **циклическое (итерационное) выполнение** (тег `loop`). Защитное условие появляется в вершине одной линии жизни внутри тела. Тело оператора цикла выполняется неоднократно – до тех пор, пока защитное условие истинно перед каждой итерацией. Когда же оно принимает значение `false` (ложь) в вершине тела, управление передается за пределы оператора.

Существует также множество других операторов, но перечисленные используются наиболее часто.

Для четкого обозначения границ диаграмма последовательности может заключаться в прямоугольник с тегом `sd` в верхнем левом углу. За этим тегом может следовать имя диаграммы.

На рис. 19.3 представлен упрощенный пример, который иллюстрирует использование некоторых управляющих операторов. Пользователь инициирует последовательность. Первый оператор – оператор

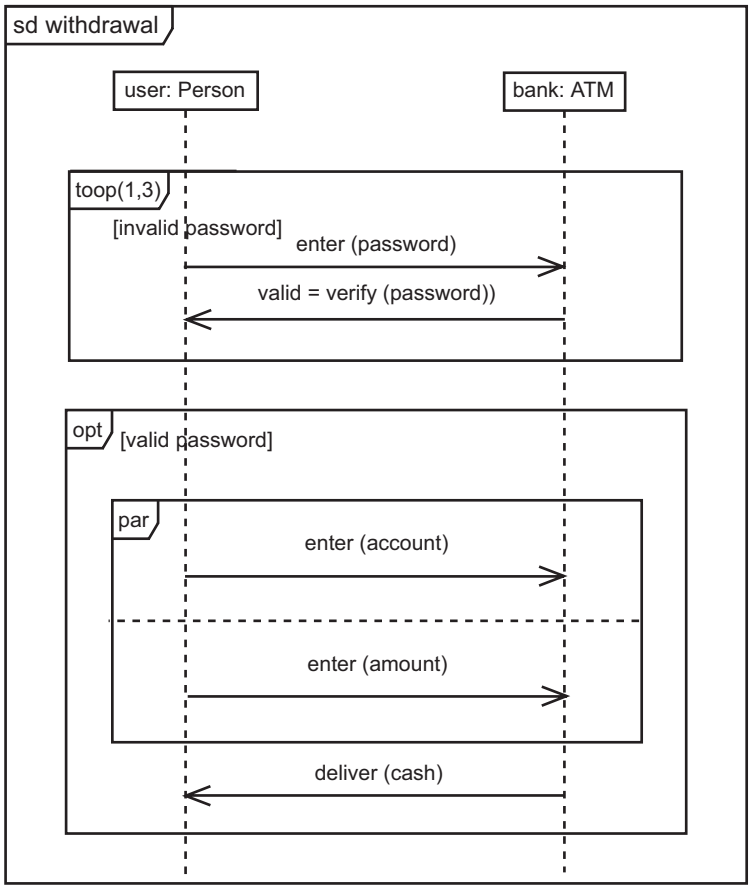


Рис. 19.3. Операторы структурированного управления

цикла. Цифры в скобках (1, 3) означают минимальное и максимальное количество выполнений тела цикла. Поскольку минимальное число – единица, это тело выполнится как минимум один раз, прежде чем будет проверено защитное условие. В цикле пользователь вводит пароль, и система проверяет его. Цикл прерывается после трех попыток, хотя может завершиться и раньше в случае ввода неправильного пароля.

Следующий оператор необязательный. Его тело выполняется, если введен правильный пароль; в противном случае остаток диаграммы последовательности пропускается. Тело необязательного оператора содержит в себе параллельный оператор. Операторы могут быть вложены, как показано на рис. 19.3.

Параллельный оператор имеет две подобласти: одна разрешает пользователю ввести номер счета, а другая – сумму. Поскольку они параллельны, для этих двух элементов не предусматривается определенный порядок ввода. Это говорит о том, что параллельность не всегда означает одновременное выполнение. На самом деле это говорит о том, что два действия не скоординированы и могут совершаться в любом порядке. Если они действительно независимы, то могут перекрывать друг друга; если же последовательны, то одно начинается по завершении другого в произвольной очередности.

После того как оба действия выполнены, параллельный оператор завершен. Далее внутри необязательного оператора банк выдает наличные пользователю. Этим исчерпывается роль диаграммы последовательности.

Вложенные диаграммы деятельности

Слишком большие диаграммы деятельности порой сложны для понимания. Поэтому их структурированные разделы могут быть организованы в виде подчиненных деятельностей – особенно в случаях, когда таковые выполняются не один раз в пределах главной. При этом главная и подчиненные деятельности изображаются на отдельных диаграммах. Внутри главной диаграммы деятельности подчиненная деятельность представлена в виде прямоугольника с тегом ref в левом верхнем углу. Имя подчиненного поведения указывается в центре рамки. Подчиненное поведение не ограничивается диаграммой деятельности; оно также может быть автоматом, диаграммой последовательности или другой поведенческой спецификацией. На рис. 19.4 показана диаграмма с рис. 19.3, перерисованная таким образом, что две секции перемещены в отдельные диаграммы деятельности, а на главной указаны ссылки на них.

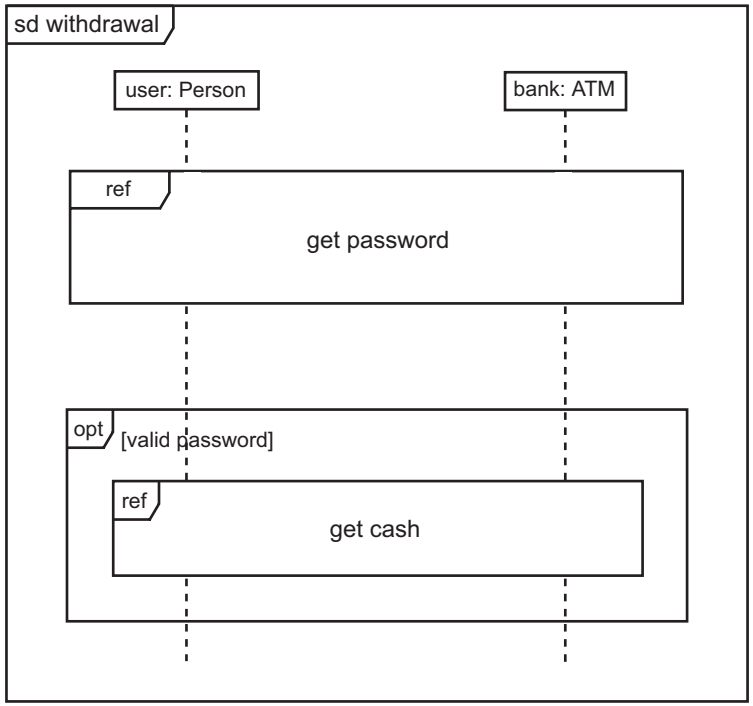


Рис. 19.4. Вложенная диаграмма деятельности

Диаграммы коммуникации

Диаграммы коммуникации описывают организацию объектов, участвующих во взаимодействии. Как показывает рис. 19.5, диаграмма коммуникации формируется начиная с размещения объектов, участвующих во взаимодействии, в вершинах графов. Далее в виде дуг графа изображаются ссылки, которые соединяют эти объекты. Связям могут быть присвоены имена ролей, идентифицирующие их. И наконец, связи дополняются сообщениями, которыми обмениваются объекты. Все это дает читателю четкое визуальное представление потока управления в контексте структурной организации взаимодействующих объектов.

На заметку. В отличие от диаграммы последовательности, на диаграмме коммуникации не продемонстрированы явно линии жизни объектов, хотя можно показать сообщения create и destroy. К тому же явно не отображается фокус управления, хотя последовательный номер каждого сообщения может указывать уровень вложенности.

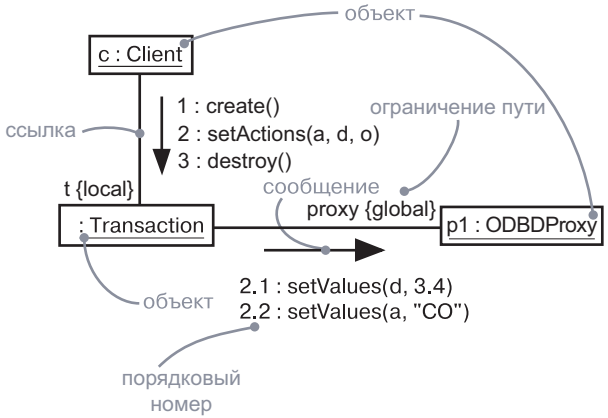


Рис. 19.5. Диаграмма коммуникации

Можно использовать расширенную форму порядковых номеров, чтобы выделить параллельные потоки управления (см. главу 23). Стереотипы путей обсуждаются в главе 18; сложное ветвление и итерации проще специфицировать на диаграммах деятельности, которые рассматриваются в главе 20.

Диаграммы коммуникации обладают двумя признаками, которые отличают их от диаграмм последовательности. Во-первых, здесь отмечен *путь* (path), который отображается соответственно ассоциации. Впрочем, можно также показать его в соответствии с локальными переменными, параметрами, глобальными переменными и обращениями к самому себе. Путь представляет источник информации для объекта.

Во-вторых, здесь имеется порядковый номер. Чтобы указать порядок сообщения во времени, вы предваряете его номером начиная с 1 и далее – в арифметической прогрессии для каждого нового сообщения в потоке управления (2, 3, ...). Чтобы показать вложенность, применяется десятичная система классификации Дьюи (1 – первое сообщение, которое содержит в себе сообщения 1.1, 1.2 и т.д.). Количество уровней вложения не ограничивается. Кроме того, следует отметить, что на линии одной и той же ссылки можно показать множество сообщений (возможно, пересылаемых в разных направлениях), и каждое будет иметь уникальный порядковый номер.

Большую часть времени вам придется моделировать прямые последовательные потоки управления. Но существует и возможность моделирования более сложных потоков, включающих в себя итерации и ветвление. *Итерация* (iteration) представляет собой повторяющуюся последовательность сообщений. Чтобы смоделировать ее, надо снабдить порядковый номер выражением итерации в формате `*[i := 1..n]` (или просто `*`, если нужно показать наличие итерации, не вдаваясь в детали). Итерация говорит о том, что сообщение и все его вложения должны повториться в соответствии с заданным выражением. Аналогичным образом условие представляет сообщение, исполнение которого участвует в вычислении

булева выражения. Чтобы смоделировать условие, порядковый номер предваряют условным выражением, таким как, например, `[x > 0]`. Альтернативный путь *ветвления* (branching) имеет тот же порядковый номер, но каждый путь должен уникально отделяться неперекрывающимися условиями.

Как для итераций, так и для ветвления UML не регламентирует формат выражения внутри скобок; вы можете использовать псевдокод либо синтаксис определенного языка программирования.

На заметку. На диаграммах последовательности связи между объектами не показываются явно. Также на них не отображаются порядковые номера сообщений; они заданы неявно физическим упорядочением сообщений на диаграмме сверху вниз. Однако допускается отображение итераций и ветвления с использованием управляющих структур диаграмм последовательности.

Семантическая эквивалентность

Поскольку и диаграммы последовательности, и диаграммы коммуникации наследуют одну и ту же информацию метамодели UML, они семантически эквивалентны. В результате вы можете взять одну диаграмму и преобразовать ее в другую без каких бы то ни было потерь информации, как можно заметить по рис. 19.2 и 19.5, которые семантически эквивалентны. Однако это не значит, что обе диаграммы явно визуализируют одну и ту же информацию. Например, диаграмма коммуникации на рис. 19.5 показывает, как связаны объекты (обратите внимание на аннотации `{local}` и `{global}`), а соответствующая диаграмма последовательности (рис. 19.2) – нет. Зато диаграмма последовательности показывает возврат сообщения (отметим возвращаемое значение `committed`), а соответствующая диаграмма коммуникации – нет. В основе обеих диаграмм лежит одна и та же модель, но представлена она несколько по-разному. Поскольку модель, описанная в одном формате, не содержит информацию, которая присутствует в другом, можно сказать, что диаграммы последовательности и коммуникации все же создают две принципиально разные модели, хотя и используют одну общую в качестве базовой.

Общее применение

Диаграммы взаимодействия применяются для моделирования динамических аспектов системы. Эти аспекты могут подразумевать взаимодействие экземпляров любого рода на любом представлении

Пять представлений архитектуры обсуждаются в главе 2, экземпляры – в главе 13, классы – в главах 4 и 9, активные классы – в главе 23, интерфейсы – в главе 11, компоненты – в главе 15, узлы – в главе 27, системы и подсистемы – в главе 32, операции – в главах 4 и 9, варианты использования – в главе 17, кооперации – в главе 28.

архитектуры системы, включая классы (в том числе активные), интерфейсы, компоненты и узлы.

Когда диаграммы взаимодействия используются для моделирования некоторых динамических аспектов системы, это делается в контексте системы в целом, подсистемы, операции либо класса. Можно также присоединить диаграммы взаимодействия к вариантам использования, чтобы моделировать сценарии, и к кооперациям, чтобы моделировать динамические аспекты сообществ объектов.

Диаграммы взаимодействия обычно используются двумя способами:

1. *Чтобы моделировать потоки управления, упорядоченные по времени*. Для этого применяются диаграммы последовательности. Моделирование потока управления, упорядоченного по времени, выделяет передачи сообщений в хронологическом порядке, что, в частности, удобно для визуализации динамического поведения в контексте сценариев вариантов использования. Диаграммы последовательности лучше выполняют задачу визуализации простых итераций и ветвления, чем диаграммы коммуникаций.
2. *Чтобы моделировать потоки управления по организации*. Для этого применяются диаграммы коммуникации. Моделирование потока управления по организации выделяет структурные связи между экземплярами во взаимодействии наряду с сообщениями, которые могут между ними передаваться.

Типичные приемы моделирования

Моделирование потоков управления, упорядоченных по времени

Рассмотрите объекты, которые находятся в контексте системы, подсистемы, операции или класса, а также объекты и роли, которые участвуют в варианте использования или кооперации. Чтобы смоделировать поток управления, проходящий через эти объекты и роли, используйте диаграммы взаимодействия; чтобы подчеркнуть временной порядок сообщений, применяйте разновидность диаграмм взаимодействий – диаграмму последовательности.

Чтобы смоделировать поток управления во времени, необходимо:

Системы и подсистемы обсуждаются в главе 32, операции и классы – в главах 4 и 9, варианты использования – в главе 17, кооперации – в главе 28.

- Установить контекст взаимодействия (то есть определить, идет ли речь о контексте системы, подсистемы, операции или класса, либо одного из сценариев варианта использования, либо кооперации).
- Установить основу взаимодействия, идентифицировав объекты, которые играют роль во взаимодействии. Расположить их на диаграмме последовательности слева направо – от более важных к зависимым.
- Изобразить линию жизни каждого объекта. Большинство из них существует на протяжении всего взаимодействия. Для тех же, которые создаются и уничтожаются во время взаимодействия, установите линии жизни соответствующим образом, явно обозначив «рождение» и «смерть» объекта сообщениями со стереотипами.
- Начиная с сообщения, которое инициирует взаимодействие, расположите все последующие сверху вниз между линиями жизни, показывая свойства каждого сообщения (в частности, его параметры), насколько это необходимо для объяснения семантики взаимодействия.
- Если вам нужно визуализировать вложенность сообщений или момента времени, в который выполняется конкретное вычисление, снабдите линию жизни каждого объекта его фокусом управления.

Временные метки обсуждаются в главе 24, пред- и постусловия – в главе 9, пакеты – в главе 12.

- Если требуется специфицировать временные или пространственные ограничения, снабдите каждое сообщение временной меткой и присоедините соответствующие ограничения времени и пространства.
- Если необходимо специфицировать поток управления более формально, подключите к каждому сообщению пред- и постусловия.

Отдельная диаграмма последовательности может показать только один поток управления (хотя допускается выражение структурной параллельности средствами управляющих структурных операторов). Обычно приходится иметь дело со множеством диаграмм взаимодействия; некоторые из них первичны, а другие показывают альтернативные пути или исключительные условия. Для организации этих наборов диаграмм последовательностей можно использовать пакеты, присваивая каждой диаграмме соответствующее уникальное имя.

В качестве примера на рис. 19.6 показана диаграмма последовательности, описывающая поток управления, который инициирует простой телефонный звонок. На этом уровне абстракции существуют четыре роли, вовлеченные в процесс: два абонента (`Callers`) – `s` и `r`, безымянный телефон `Switch` и экземпляр `c` класса `Conversation`

(Разговор) между двумя абонентами. При том что диаграмма моделирует четыре роли, каждый экземпляр диаграммы имеет конкретные объекты, связанные с каждой из ролей. Один и тот же образец взаимодействия применяется к каждому экземпляру диаграммы.

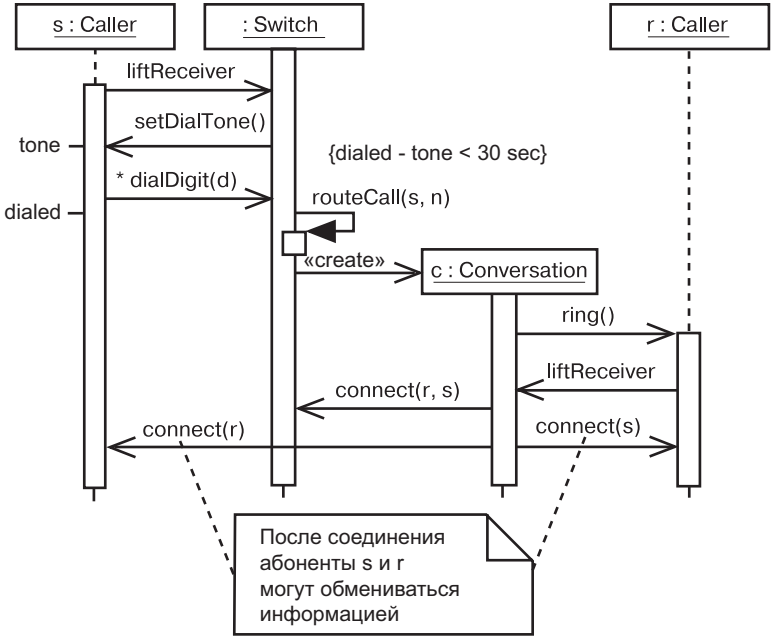


Рис. 19.6. Моделирование потока управления во времени

Последовательность начинается с того, что один Caller (абонент s) посылает сигнал (liftReceiver – снятиеТрубки) объекту Switch. В свою очередь, Switch посылает setDialTone (длинныйГудок) объекту Caller, который начинает итерацию сообщений dialDigit (наборНомера). Ограничение показывает, что эта последовательность должна занимать не дольше 30 с. Диаграмма ничего не сообщает о том, что случится, если временное ограничение будет нарушено. Чтобы показать последствия этого события, можно включить ветвь или же отдельную диаграмму последовательности. Затем объект Switch вызывает самого себя для выполнения операции routeCall (передатьВызов). Далее он создает объект Conversation (c), которому поручает всю остальную работу. Хотя здесь это и не показано, объект c должен выполнять дополнительные обязанности как часть механизма, учитывающего стоимость разговоров и выставляющего счета абонентам (данный механизм должен быть описан в другой диаграмме взаимодействия). Объект Conversation (c) звонит Caller (r), который асинхронно посылает сообщение liftReceiver. Затем объект Conversation говорит

Switch, чтобы тот вызвал операцию connect (соединить), и сообщает обоим абонентам, что нужно выполнить connect, после чего они смогут обмениваться информацией, как показано в примечании.

Диаграмма взаимодействия может начинаться или заканчиваться в любой точке последовательности. Полная трассировка потока управления может быть чрезвычайно сложной, поэтому разумно выделить объемные потоки в отдельные диаграммы.

Моделирование потоков управления по организации

Системы и подсистемы обсуждаются в главе 32, операции и классы – в главах 4 и 9, варианты использования – в главе 17, кооперации – в главе 28.

Рассмотрите объекты, существующие в контексте системы, подсистемы, операции или класса, а также объекты и роли, принимающие участие в кооперации. Чтобы смоделировать поток управления, который проходит через эти объекты и роли, используйте диаграмму взаимодействия; чтобы показать передачу сообщений в контексте структуры, используйте разновидность диаграмм взаимодействия – диаграмму коммуникации.

Чтобы смоделировать поток управления по организации, необходимо:

- Установить контекст взаимодействия (то есть определить, идет ли речь о контексте системы, подсистемы, операции или класса, либо одного из сценариев варианта использования, либо кооперации).
- Установить основу взаимодействия, идентифицировав объекты, которые играют роль во взаимодействии. Расположить их на диаграмме коммуникации в вершинах графа, помещая наиболее важные объекты в центре диаграммы, а вторичные – ближе к краям.

Связи зависимости обсуждаются в главах 5 и 10, ограничения, связанные с путями, – в главе 16.

- Специфицировать ссылки между объектами наряду с пересылаемыми по ним сообщениями:

  – сначала расположить ссылки ассоциаций – они наиболее важны, потому что представляют структурные соединения;
  – потом изобразить остальные связи и снабдить их соответствующими путевыми аннотациями (такими как global или local), чтобы явно показать, как эти объекты относятся друг к другу.

- Начиная с сообщения, инициирующего взаимодействие, присоединить каждое последующее сообщение к соответствующей ссылке, указывая соответствующий порядковый номер. Вложенность сообщений показать в нотации десятичной системы классификации Дьюи.

Временные метки обсуждаются в главе 24, пред- и постусловия – в главе 4, пакеты – в главе 12.

- Если необходимо указать временные и пространственные ограничения, снабдить каждое сообщение временной меткой и присоединить нужное временное или пространственное ограничение.
- Если требуется описать поток управления более формально, присоединить к каждому сообщению пред- и постусловие.

Как и диаграммы последовательности, диаграмма коммуникации может показать только один поток управления (хотя некоторые простые вариации можно отобразить в нотации взаимодействий и ветвления UML). Обычно приходится иметь дело со множеством таких диаграмм взаимодействия, одни из которых первичны, а другие показывают альтернативные пути или исключительные условия. Также вы можете использовать пакеты для организации этих наборов диаграмм коммуникации, присваивая каждой из них уникальное имя.

На рис. 19.7 в качестве примера показана диаграмма коммуникации, описывающая поток управления в процессе регистрации нового студента в учебном заведении, с подчеркиванием структурных связей между этими объектами. Здесь вы видите четыре роли: `RegistrarAgent` (АгентРегистрации – `r`), `Student` (Студент – `s`), `Course` (Курс – `c`) и безымянную роль `School` (Учебное заведение). Поток управления явно пронумерован. Действие начинается с того, что `RegistrarAgent` создает объект `Student`, добавляя его к учебному заведению (сообщение `addStudent` – добавитьСтудента), а затем извещая объект `Student` о том, что ему нужно зарегистрироваться. Последний вызывает операцию `getSchedule` (узнатьРасписание) и получает перечень объектов `Course` – курсов, на которых он должен зарегистрироваться. Объект `Student` добавляет себя к каждому объекту `Course` в этом наборе.

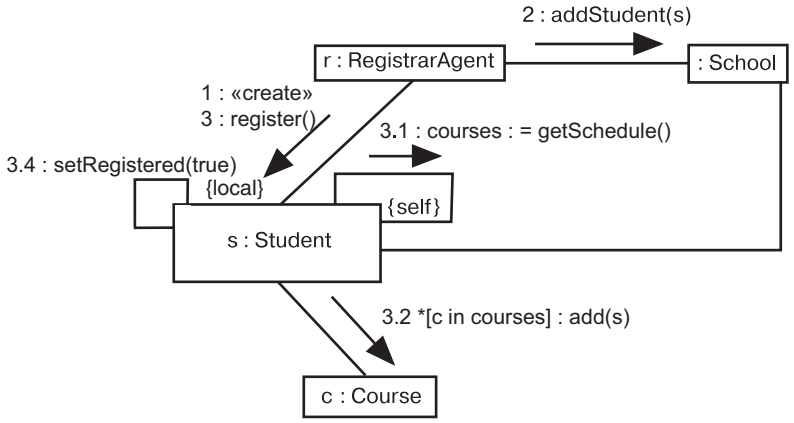


Рис. 19.7. Моделирования потока управления по организации

Прямое и обратное проектирование

Прямое проектирование (создание кода из модели) возможно и для диаграммы последовательности, и для диаграммы коммуникации, особенно если контекстом таковой является операция. Например, применяя предыдущую диаграмму коммуникации, достаточно «умное» инструментальное средство может сгенерировать следующий код операции `register` (регистрировать), включенной в класс `Student` на языке Java:

```
public void register() {
  CourseCollection courses = getSchedule();
  for(int i = 0; i < courses.size(); i++)
    courses.item(i).add(this);
  this.registered = true;
}
```

«Достаточно умное средство» – такое, которое может реализовать метод `getSchedule`, возвращающий объект `CourseCollection` (СписокКурсов), определяемый на основе сигнатуры операции. Проходя по содержимому объекта с применением стандартной идиомы итераций (о которой, безусловно, инструмент должен знать), код затем обобщается для любого количества предложенных курсов.

Обратное проектирование (воссоздание модели из кода) также возможно и для диаграмм последовательности, и для диаграмм коммуникации, особенно если речь идет о контексте операции. Сегменты предыдущей диаграммы могли бы быть восстановлены инструментом из прототипа операции `register`.

На заметку. Прямое проектирование просто, обратное сложно. При типичном обратном проектировании иной раз появляется слишком много информации, чтобы можно было с легкостью решить, какие детали следует сохранить, а какие – отбросить.

Однако более интересной, чем обратное проектирование модели из кода, может быть анимация модели на основе работающей системы. Например, в случае с предыдущей диаграммой инструмент может анимировать сообщения, показывая, как они передаются в работающей системе. Более того, имея такой инструмент под управлением отладчика, вы можете управлять скоростью выполнения – возможно, устанавливая точки прерывания, чтобы приостановить процесс в интересных местах и просмотреть значения атрибутов конкретных объектов.

Советы и подсказки

Когда вы создаете диаграммы взаимодействия в UML, помните, что диаграммы последовательности и коммуникации являются проекциями одной и той же модели динамических аспектов системы. Ни одна диаграмма взаимодействия в отдельности не может охватить все динамические аспекты системы, а также ее подсистем, операций, классов, вариантов использования и кооперации.

Хорошо структурированная диаграмма взаимодействия наделена следующими свойствами:

- сосредоточена на передаче лишь одного аспекта динамики системы;
- включает в себя лишь те элементы, которые существенны для понимания данного аспекта;
- представляет уровень детализации, соответствующий уровню абстракции, и включает лишь дополнения, существенные для понимания диаграммы;
- не настолько лаконична, чтобы создать у читателя неверное представление о существенной семантике.

Когда вы строите диаграмму взаимодействия:

- присваивайте ей имя, соответствующее ее назначению;
- используйте диаграмму последовательности, если нужно подчеркнуть порядок сообщений во времени. Используйте диаграмму коммуникации, если хотите сделать акцент на организации объектов, участвующих во взаимодействии;
- размещайте элементы так, чтобы пересекающихся линий было как можно меньше;
- используйте примечания и выделения цветом, чтобы привлечь внимание к важным деталям диаграмм;
- не злоупотребляйте ветвлением. Вы можете гораздо лучше изобразить сложные ветвления на диаграммах деятельности.

Глава 20. Диаграммы деятельности

В этой главе:

Помимо диаграмм деятельности, динамические аспекты систем моделируют диаграммы последовательности, коммуникации, состояний и вариантов использования. Диаграммы последовательности и коммуникации обсуждаются в главе 19, диаграммы состояний – в главе 25, диаграммы вариантов использования – в главе 18. Действия рассматриваются в главе 16.

Диаграммы деятельности – это один из пяти видов диаграмм, применяемых в UML для моделирования динамических аспектов систем. По сути, диаграмма деятельности представляет собой блок-схему, которая показывает, как поток управления переходит от одной деятельности к другой. В отличие от традиционной блок-схемы диаграмма деятельности показывает параллелизм так же хорошо, как и ветвление потока управления.

Моделирование динамических аспектов систем при помощи диаграмм деятельности большей частью подразумевает моделирование последовательных (а иногда и параллельных) шагов вычислительного процесса. Кроме того, с помощью диаграмм деятельности можно моделировать поток передачи данных от одного шага процесса к другому. Диаграммы деятельности могут использоваться отдельно для визуализации, специфицирования, конструирования и документирования динамики сообщества объектов либо для моделирования потока управления в операции. Если в диаграммах взаимодействия акцент делается на переходы потока управления от одного объекта к другому, то диаграммы деятельности описывают переходы потока управления от одного шага процесса к другому. *Деятельность* – это структурированное описание текущего поведения. Осуществление деятельности в конечном счете раскрывается в виде выполнения отдельных действий, каждое из которых может изменять состояние системы или передавать сообщения.

Диаграммы деятельности важны не только для моделирования динамических аспектов системы, но и для конструирования исполняемых систем посредством прямого и обратного проектирования.

Введение

Рассмотрим поток работ (workflow) при строительстве дома. Сначала выбирается место под застройку. Затем вы нанимаете архитектора, который проектирует дом. После согласования проекта застройщик составляет смету для оценки общих затрат. Как только вы принимаете план и цену, начинается строительство. Власти дают разрешение, роется котлован, заливается фундамент, возводится каркас и т.д., пока вся работа не будет завершена. Наконец, вам вручают ключи и документы, подтверждающие право проживания, и вы вступаете во владение жильем.

Хотя на самом деле процесс строительства намного сложнее, вышеприведенное описание дает представление об основных операциях. В реальных проектах множество разных процессов можно вести параллельно: например, электрики работают одновременно с водопроводчиками и столярами. Кроме того, вы обнаружите в этих процессах условия и ветвление. Скажем, в зависимости от качества грунта вам придется либо выполнять взрывные работы, либо копать, либо размывать почву. Могут также присутствовать итерации: например, строительная инспекция обнаружила какое-то серьезные нарушения, в результате чего приходится ломать и переделывать часть здания.

В строительной промышленности для визуализации, специфицирования, конструирования и документирования рабочего процесса обычно применяются диаграммы (графики) Гантта и Перта.

Моделирование структурных аспектов системы обсуждается в частях II и III, диаграммы взаимодействия рассматриваются в главе 19. Действия обсуждаются в главе 16

При разработке программных систем возникает сходная проблема: как лучше всего смоделировать такие динамические аспекты системы, как поток работ или операцию? И здесь у вас есть два основных пути, аналогичных использованию диаграмм Гантта и Перта.

С одной стороны, вы можете построить раскадровку сценариев, которые включают взаимодействие ряда интересующих вас объектов и передаваемых между ними сообщений. Эти раскадровки в UML можно моделировать двумя способами: подчеркивая временной порядок сообщений на диаграммах последовательности либо выделяя структурные связи между взаимодействующими объектами на диаграммах коммуникации. Такие диаграммы взаимодействия чем-то сродни графикам Гантта, которые сосредоточены на объектах (ресурсах), обеспечивающих некоторую деятельность в течение определенного времени.

С другой стороны, эти же динамические аспекты можно моделировать с помощью диаграмм деятельности, которые сосредотачивают внимание в первую очередь на деятельности, в которую вовлечены объекты (см. рис. 20.1). Такие диаграммы подобны графикам Перта. По сути, диаграмма деятельности – это блок-схема, отображающая деятельность, развертывающуюся во времени. Диаграмму деятельности можно представить как диаграмму взаимодействия, «вывернутую наизнанку». Диаграмма взаимодействия рассматривает объекты, через которые проходят сообщения; диаграмма деятельности рассматривает операции, выполняемые между объектами. Семантическое различие весьма тонкое, но в результате мы получаем принципиально различные взгляды на мир.

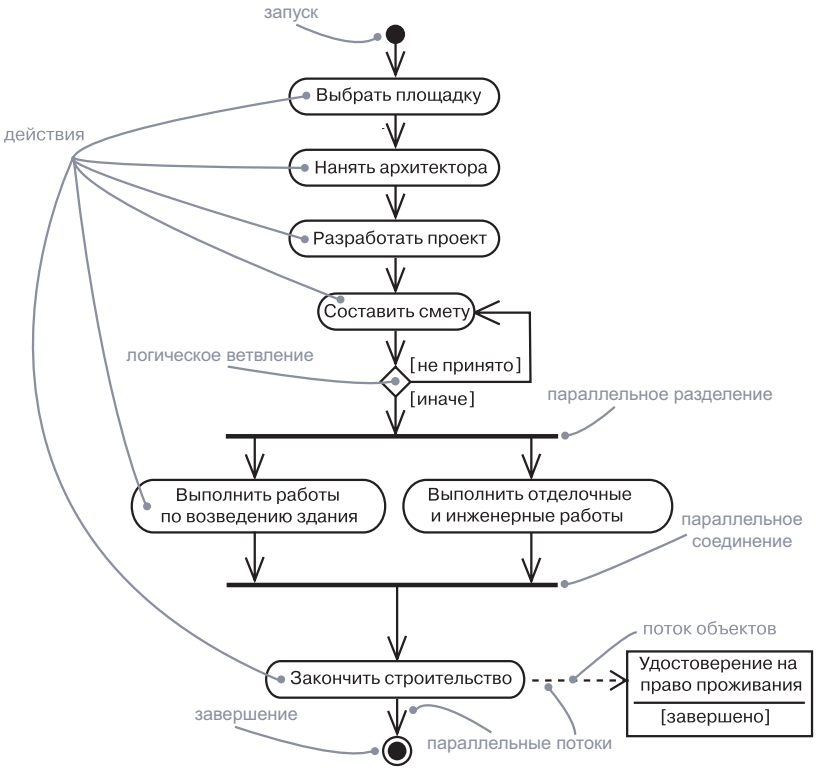


Рис. 20.1. Диаграмма деятельности

Базовые понятия

Диаграмма деятельности показывает поток управления – от одной деятельности к другой.

Деятельность (activity) – это выполняющийся в данный момент неатомарный набор действий внутри машины состояний (автомата). Выполнение некоторой деятельности в конечном счете раскрывается в виде выполнения отдельных *действий* (actions), каждое из которых может изменить состояние системы или передавать сообщения. Действия заключаются в вызове другой операции, посылке

сигнала, создании или уничтожении объекта либо в выполнении простых вычислений (например, вычислении выражения). Диаграмма деятельности представляет собой набор узлов и дуг.

Общие свойства

Общие свойства диаграмм обсуждаются в главе 7.

Диаграмма деятельности обладает рядом свойств, общих для всех диаграмм: она наделена именем и графическим наполнением, являющимся проекцией модели. От диаграмм других типов отличается своим содержимым.

Содержимое

Состояния, переходы и машины состояний (автоматы) обсуждаются в главе 22, объекты – в главе 13.

Диаграммы деятельности обычно содержат действия, узлы деятельности, потоки и значения объектов. Как и все прочие диаграммы, могут содержать примечания и ограничения.

Действия и узлы деятельности

В потоке управления, моделируемом диаграммой деятельности, нечто происходит: вы можете вычислить выражение, которое устанавливает значение атрибута или возвращает некоторое значение, вызвать операцию объекта, послать ему сигнал либо даже создать или уничтожить объект. Эти исполняемые атомарные вычисления называются *действиями*. Как показано на рис. 20.2, действие изображается в виде овала. Внутри него можно написать выражение.

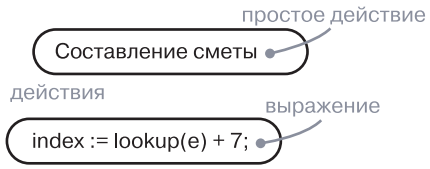


Рис. 20.2. Действия

Атрибуты и операции обсуждаются в главах 4 и 9, сигналы – в главе 21, создание и уничтожение объектов – в главе 16, состояния и автоматы – в главе 22.

На заметку. UML не предусматривает языка для подобных выражений. Абстрактно можно использовать структурированный текст, а более конкретно – синтаксис и семантику определенного языка программирования.

Действия не могут быть подвергнуты декомпозиции. Более того, они атомарны – это означает, что события могут происходить, но внутреннее поведение действия невидимо. Вы не можете выполнить часть действия: оно либо выполняется целиком, либо не выполняется вообще. Наконец, часто предполагается, что время выполнения действий несущественно, хотя некоторые из них могут требовать ощутимых затрат времени.

Моделирование времени и пространства обсуждается в главе 24.

На заметку. В реальном мире, конечно, каждое вычисление требует времени и пространства (особенно это важно для систем реального времени). Важно иметь возможность смоделировать такие свойства.

Узел деятельности (activity node) – это организационная единица деятельности. Вообще, он представляет собой вложенную группу действий или других вложенных узлов. Более того, узлы деятельности имеют видимую подструктуру и, как правило, требуют некоторого времени на завершение. Действие можно представлять как частный случай узла деятельности. Точнее говоря, это такой узел деятельности, который не может быть подвергнут декомпозиции. Аналогичным образом узел деятельности можно рассматривать как композицию, поток управления которой состоит из других действий и узлов. Если погрузиться в детали внутреннего устройства узла деятельности, там можно обнаружить другую диаграмму деятельности. Как показано на рис. 20.3, нотация действия и узла деятельности одинакова, за исключением того что последний может иметь дополнительные части, которые обычно поддерживаются инструментом редактирования за пределами диаграммы.

Рис. 20.3. Узлы деятельности

Потоки управления

Когда некоторое действие или узел деятельности завершает выполнение, поток управления немедленно переходит к следующему действию или узлу деятельности. Этот поток изображается при помощи стрелочек, показывающих его путь от одного действия или узла к другому. В UML поток управления представлен в виде простой стрелочки, направленной от предшественника к последователю, без метки события (рис. 20.4).

В самом деле, поток управления где-то должен начинаться и завершаться, если только он не бесконечен. Поэтому, как показано на рис. 20.4, вы можете выразить специальными символами инициализацию (круг) и завершение (круг, заключенный в окружность).

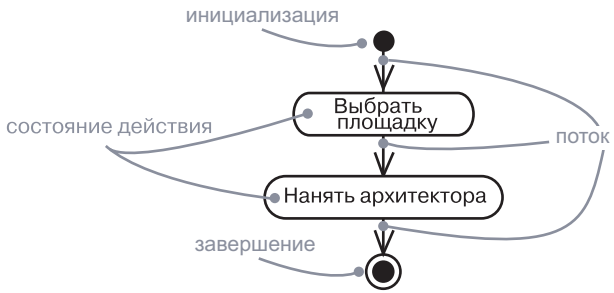


Рис. 20.4. Завершение переходов

Ветвление

Ветвление – это соглашение об обозначении, семантически эквивалентное множественным переходам с защитными условиями (см. главу 22).

Простые последовательные потоки встречаются чаще всего, но это не единственное средство моделирования потока управления. Так же, как в блок-схеме, можно включать в диаграмму ветви, которые специфицируют альтернативные пути, выбираемые на основе булевых выражений. *Ветвление* (branching) может иметь один входящий поток и несколько исходящих. На каждом исходящем потоке помещается булево выражение условия, которое вычисляется на входе в ветвь. Условия выходных потоков не должны перекрываться (в противном случае поток управления будет неоднозначным), но при этом должны учитывать все возможные варианты (в противном случае поток управления может оказаться «замороженным»). Как показано на рис. 20.5, ветвление изображается ромбиком.

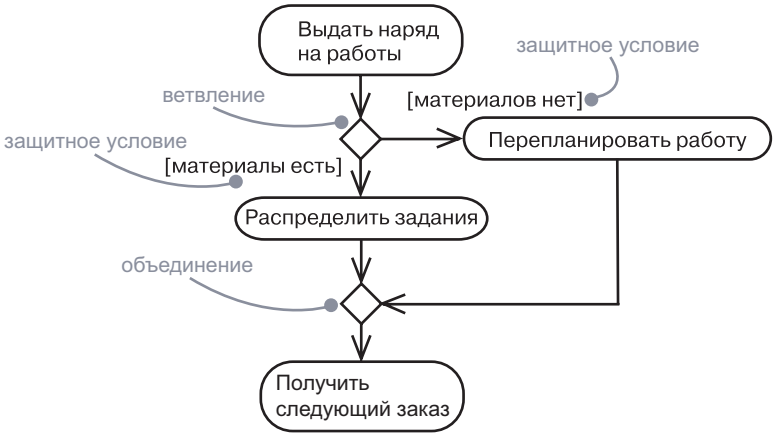


Рис. 20.5. Ветвление

Для удобства можно использовать ключевое слово `else`, чтобы пометить исходящую ветвь потока, которая выполняется, если не истинно ни одно из других выражений.

Ветвление и итерации можно изобразить на диаграммах взаимодействия (см. главу 19).

Когда два пути потока управления вновь сходятся вместе, можно задействовать специальный символ – ромбик с двумя входящими стрелочками и одной исходящей. Для объединения потоков никаких условий не требуется.

Можно достичь эффекта итерации, используя одно действие, устанавливающее значение итератора, и ветвление, которое вычисляет признак завершения итерации. UML предусматривает типы узлов для циклов, но это зачастую легче выразить текстовыми, а не графическими средствами.

На заметку. UML не предусматривает языка для ветвления и итерации. На абстрактном уровне можно использовать структурированный текст, на более конкретном – синтаксис и семантику определенного языка программирования.

Разделение и соединение

Параллельный поток управления часто существует в контексте независимого активного объекта, который обычно моделируется как процесс или поток (см. главу 23). Узлы описываются в главе 27.

Простые и ветвящиеся последовательные переходы встречаются на диаграммах деятельности чаще всего. Однако иногда, особенно при моделировании бизнес-процессов, могут встретиться и параллельные потоки. В UML для описания разделения и соединения параллельных потоков управления применяют *линейку синхронизации* (synchronization bar). Она изображается в виде жирной горизонтальной или вертикальной линии.

Рассмотрим в качестве примера параллельные потоки в системе, которая имитирует человеческую речь и жестикуляцию. Как показано на рис. 20.6, *разделение* (forking) представляет собой расщепление одного потока управления на несколько параллельных. В этом процессе присутствуют один входной и ряд независимых выходных потоков управления. После разделения деятельность, ассоциированная с каждым из них, выполняется параллельно. Концептуально деятельности в каждом потоке действительно параллельны, хотя в реальной работающей системе они могут быть не только параллельными (как в случае, если система развернута на нескольких узлах), но и состоящими из прерывистых последовательностей (когда она развернута на одном узле), что создает иллюзию параллельности.

Как следует из рис. 20.6, *соединение* (joining) представляет синхронизацию нескольких параллельных потоков управления. Соединение может иметь ряд входящих потоков и один исходящий. Перед соединением потоков деятельности, которые ассоциированы с каждым из них, выполняются параллельно; в момент соединения потоки синхронизируются, то есть теперь каждый из них ждет, когда все остальные достигнут точки соединения, начиная с которой продолжит выполняться один поток управления.

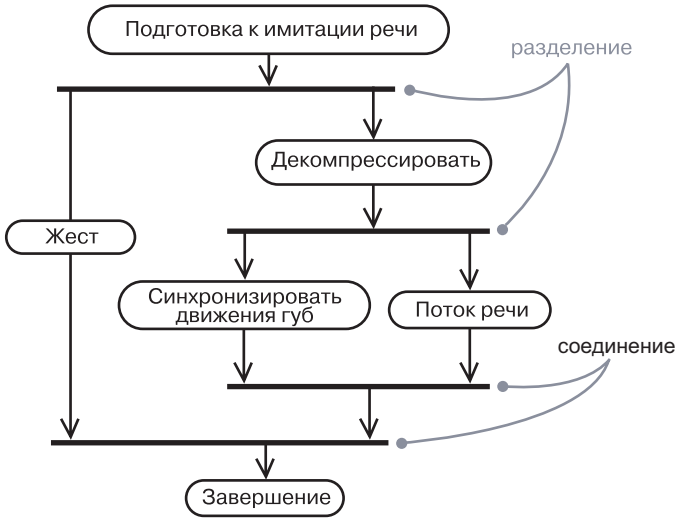


Рис. 20.6. Разделение и соединение

Активные объекты обсуждаются в главе 23, сигналы – в главе 21.

На заметку. Между точками разделения и соединения должен поддерживаться баланс. Это означает, что число потоков, исходящих из точки разделения, должно быть равно числу потоков, приходящих в соответствующую точку соединения. Деятельности, выполняемые в параллельных потоках, могут обмениваться информацией между собой, посылая сигналы. Такое средство организации взаимодействия последовательных процессов называется *сопрограммой* (coroutine). В большинстве случаев при моделировании подобного взаимодействия применяются активные объекты.

«Плавательные дорожки»

При моделировании бизнес-процессов вы обнаружите, что на диаграммах деятельности иногда желательно разбивать состояния деятельности на группы, каждая из которых представляет отдел компании, занимающийся определенной деятельностью. В UML такие группы называются *«плавательными дорожками»* (swimlanes), потому что визуально каждая группа отделяется от соседних вертикальной чертой, подобно плавательным дорожкам в бассейне (см. рис. 20.7). Дорожки определяют наборы деятельностей, которым присуще некоторое общее организационное свойство.

Дорожка – это разновидность пакета. Пакеты обсуждаются в главе 12, классы – в главах 4 и 9, процессы и потоки – в главе 23.

Каждой дорожке на диаграмме присваивается уникальное имя. На самом деле дорожка не несет никакой глубокой семантики – разве что может отражать такую сущность реального мира, как, например, организационное подразделение компании. Каждая дорожка представляет собой высокоуровневую обязанность части деятельности, отображенной на диаграмме, и в конечном счете может быть реализована в виде одного или нескольких классов. На диаграмме деятельности, разбитой на дорожки, каждая деятельность принадлежит только одной из них, но переходы могут пересекать границы дорожек.

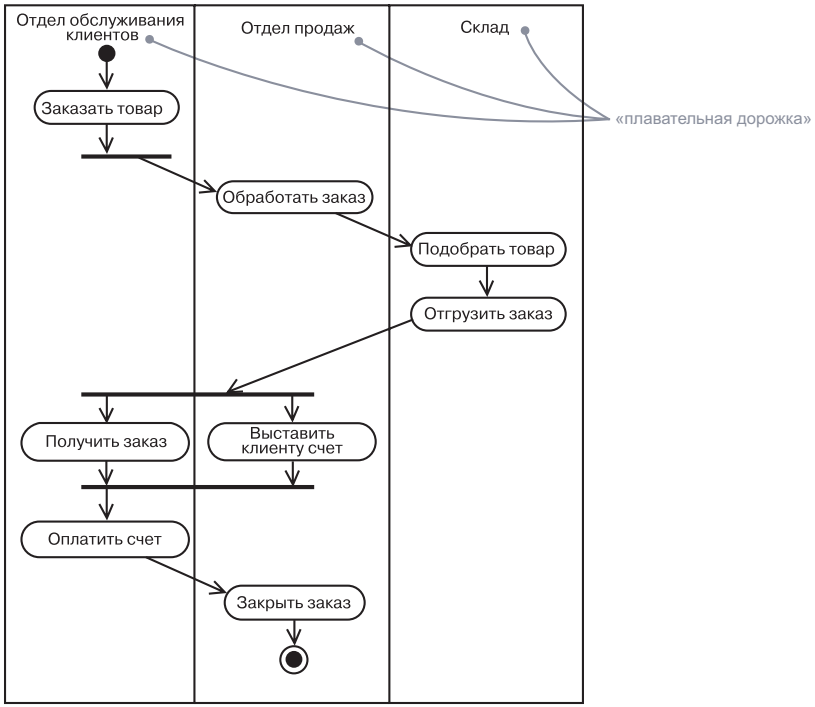


Рис. 20.7. «Плавательные дорожки»

На заметку. Существует некоторая связь между дорожками и параллельными потоками управления. Концептуально деятельность внутри каждой дорожки обычно (хотя и не всегда) рассматривается отдельно от деятельности в соседних. Это не лишено смысла, поскольку в реальном мире подразделения организации, представленные дорожками, как правило, независимы и функционируют параллельно.

Поток объектов

Объекты обсуждаются в главе 13, моделирование словаря системы – в главе 4.

Объекты могут участвовать в потоке управления, ассоциированном с диаграммой деятельности. Например, для последовательности операций обработки заказа, которая изображена на рис. 20.7, словарь проблемной области, вероятно, будет включать классы `Order` (Заказ) и `Bill` (Счет). Экземпляры этих двух классов будут созданы в результате некоторой деятельности – например, деятельность `Process order` (Обработать заказ) создаст объект `Order`), тогда, как другие виды деятельности могут использовать или модифицировать эти объекты – например, деятельность `Ship order` (Отгрузить заказ) может изменить состояние `Order` на `filled` (выполнен).

Связи зависимости обсуждаются в главах 5 и 10.

Как показано на рис. 20.8, вы можете специфицировать сущности, которые имеют отношение к диаграмме деятельности, разместив их на ней и соединив стрелочками с деятельностями, которые их создают или используют.

Такое явление называется *потоком объектов* (object flow), потому что здесь действительно имеет место поток значений объекта от одной деятельности к другой. Поток объектов, по существу, подразумевает наличие потока управления (невозможно выполнить деятельность, которая требует значения, не имея этого значения!), поэтому нет необходимости рисовать поток управления между деятельностями, соединенными потоками объектов.

Значения и состояния объекта обсуждаются в главе 13, атрибуты – в главах 4 и 9.

Помимо самого потока объекта на диаграмме деятельности вы можете показать, как изменяются его роль, состояние и значения атрибутов. По рис. 20.8 видно, что для изображения состояния объекта имя этого состояния заключается в скобки и помещается под именем объекта.

Области расширения

Часто одну и ту же операцию нужно выполнять для всех элементов набора. Например, если заказ включает в себя список позиций, обработчик заказа должен выполнить одну и ту же операцию для каждой строки этого списка: проверить наличие товара, посмотреть цену, узнать, облагается ли данная позиция налогом и т.д. Операции над списками часто моделируются в виде циклов, но при этом проектировщик должен предусмотреть проход по всем позициям, извлечь их все одну за другой, выполнить операцию, добавить результат в выходной массив, увеличить индекс и проверить цикл на завершенность. Механика выполнения цикла скрывает действительный смысл операции. Этот весьма часто применяемый образец можно смоделировать, применяя *область расширения* (expansion region).

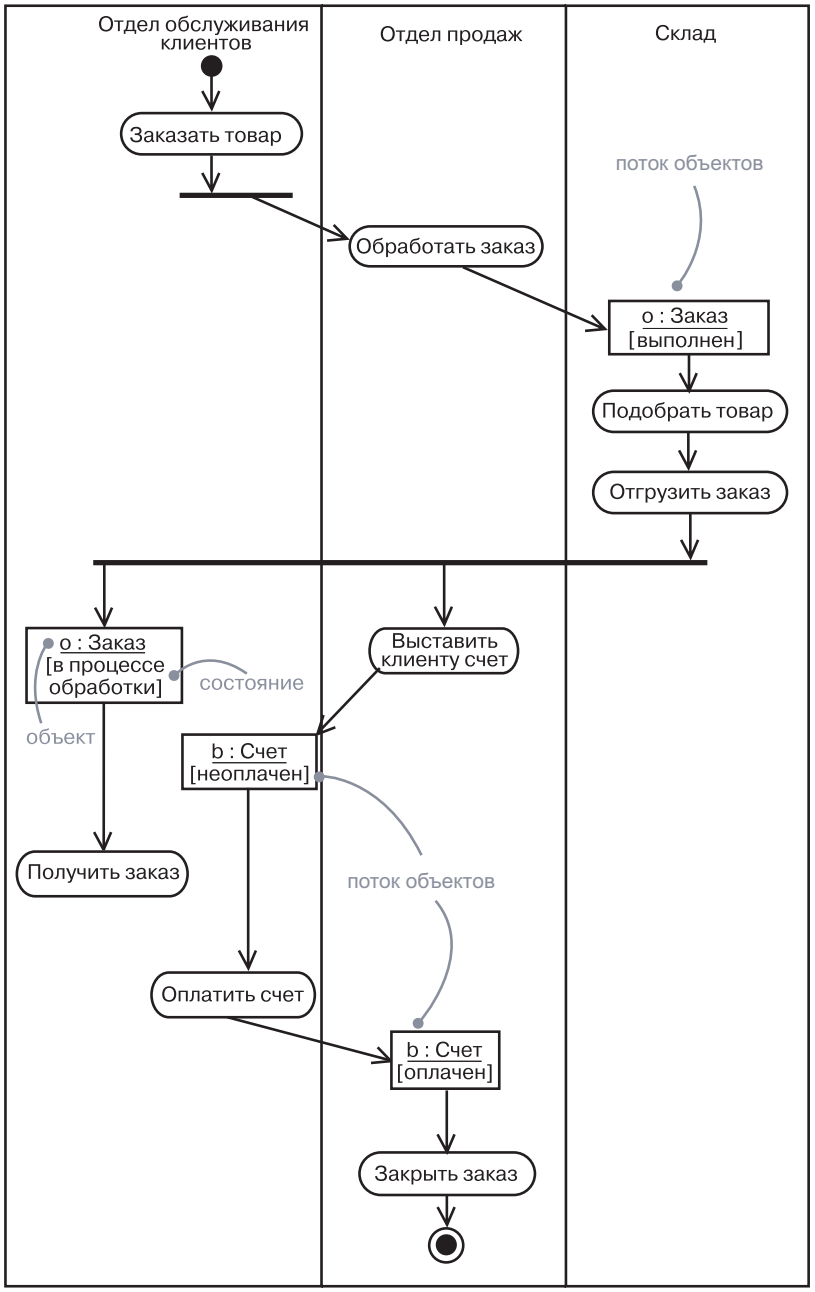


Рис. 20.8. Поток объектов

Область расширения представляет фрагмент модели деятельности, который выполняется для списка или набора элементов. На диаграмме деятельности изображается пунктирной линией, проведенной вокруг области диаграммы. Ввод в область расширения и вывод из нее представляют собой наборы значений (таких как строки списка элементов заказа). Входные и выходные наборы изображаются в виде ряда маленьких квадратиков, соединенных друг с другом; они символизируют массив значений. Когда значение массива поступает во входную коллекцию области расширения из предшествующей части модели деятельности, оно разбивается на индивидуальные элементы.

Затем область расширения исполняется для каждого элемента массива. Нет необходимости моделировать итерацию, – в области расширения она подразумевается неявно. Притом обработка нескольких элементов набора по возможности происходит параллельно. Когда завершается обработка элемента, его выходное значение, если оно есть, помещается в выходной массив в том же порядке, в котором элементы расположены во входном. Другими словами, область расширения выполняет операцию `forall` (для всех) над элементами существующего массива, чтобы создать новый.

В простейшем случае область расширения имеет один входной и один выходной массив, но в принципе у нее могут быть один или несколько входных массивов и ноль или несколько выходных. Все эти массивы должны быть одного размера, но не обязаны включать только однотипные значения. Значения из соответствующих позиций исполняются вместе для генерации выходных значений в тех же позициях. Область расширения может иметь нулевое число выходов, если все ее операции дают некий побочный эффект в отношении каждого элемента массива.

Области расширения позволяют изобразить операции над индивидуальными элементами и над их наборами в пределах одной диаграммы, без необходимости показывать все детали устройства механизма итераций.

Рис. 20.9 демонстрирует пример области расширения. В основном теле диаграммы принимается заказ. Это событие генерирует значение типа `Order` (Заказ), которое состоит из значений типа `LineItem` (СтрокаЗаказа). Значение `Order` служит входным для области расширения.

Каждое исполнение области расширения работает с одним элементом из набора `Order`. Поэтому внутри области тип входного значения соответствует одному элементу массива `Order`, а именно `LineItem`. Деятельность в пределах области расширения разделяется на две: одна находит `Product` (Продукт) и включает его в поставку, а другая вычисляет стоимость этой позиции. Необязательно обрабатывать

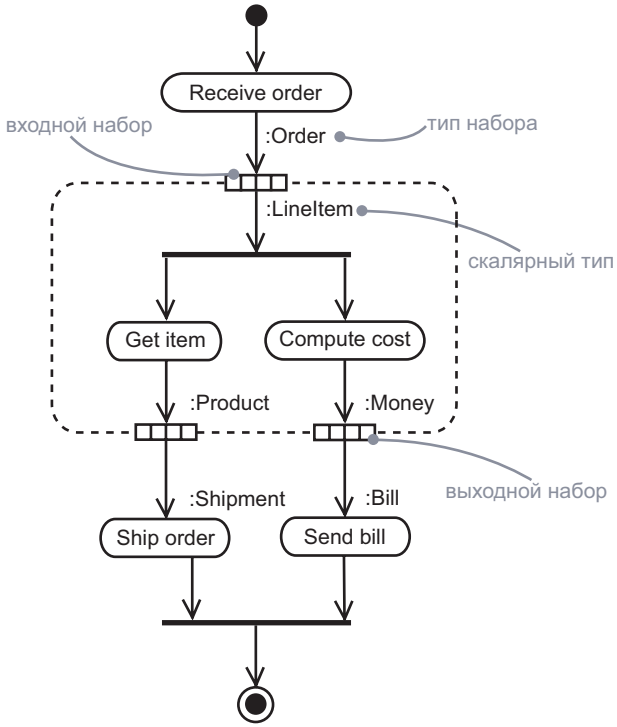


Рис. 20.9. Область расширения

элементы `LineItem` по порядку; разные выполнения области расширения могут происходить параллельно. После того как все элементы обработаны, из них формируется объект `Shipment` (Поставка) – набор продуктов, – а цены формируют `Bill` (Счет) – набор значений типа `Money` (Деньги). Значение объекта `Shipment` служит входным параметром деятельности `ShipOrder` (ОтгрузитьЗаказ), а значение объекта `Bill` – входным для деятельности `SendBill` (Выслать счет).

Общее применение

Диаграммы деятельности применяются для моделирования динамических аспектов поведения системы. Эти динамические аспекты могут включать деятельность на любом уровне абстракции в любом представлении системной архитектуры, включая классы (в том числе активные), интерфейсы, компоненты и узлы.

Использовать диаграммы деятельности для моделирования некоторых динамических аспектов системы вы можете в контексте почти любого моделируемого элемента. Однако чаще вы будете прибегать к таким диаграммам в контексте всей системы, подсистемы, операции или класса. Диаграмму деятельности можно присоединить

к варианту использования, чтобы моделировать сценарий, и к кооперации, чтобы моделировать динамические аспекты поведения совокупности объектов.

При моделировании динамических аспектов системы диаграммы деятельности обычно используются двумя способами:

1. *Для моделирования потока работ*. Внимание сосредоточено на том, как выглядит деятельность с точки зрения действующих лиц, взаимодействующих с системой. Потоки работ находятся на периферии программных систем и применяются для визуализации, спецификации, конструирования и документирования бизнес-процессов, которые включает разрабатываемая система. При таком использовании диаграмм деятельности особенное значение приобретает моделирование потоков объектов.
2. *Для моделирования операции*. В этом случае диаграммы деятельности выступают в качестве схем, моделирующих подробности вычислительного процесса. При этом особенно важно моделирование ветвления, разделения и соединения потоков управления. Контекст используемой таким образом диаграммы деятельности включает параметры операций и их локальные объекты.

Пять представлений архитектуры обсуждаются в главе 2, классы – в главах 4 и 9, активные классы – в главе 23, интерфейсы – в главе 11, операции – в главах 4 и 9, варианты использования и действующие лица – в главе 17, компоненты – в главе 15, узлы – в главе 27, системы и подсистемы – в главе 32.

Типичные приемы моделирования

Моделирование потока работ

Ни одна программная система не существует в изоляции. Всегда существует некоторый контекст, в котором она «живет» и который обязательно включает действующие лица, работающие с системой. Особенно когда речь идет о критически важном программном обеспечении масштаба предприятия, всегда можно обнаружить автоматизированные системы, работающие в контексте более высокоуровневых бизнес-процессов. Эти бизнес-процессы следует рассматривать как разновидность *потока работ* (workflow), поскольку они по своей сути представляют поток работ и объектов, проходящих через весь бизнес. Например, в розничной торговле существуют некоторые автоматизированные системы: сеть кассовых терминалов взаимодействует с системами маркетинга, складскими системами и т.д., равно как и с человеческими системами – продавцами, сотрудниками отделов маркетинга, закупок и поставок. Бизнес-процессы можно моделировать как совместную работу всех этих автоматизированных и человеческих систем, используя диаграммы деятельности.

Моделирование контекста системы обсуждается в главе 18.

Моделирование словаря системы обсуждается в главе 4, пред- и постусловия – в главе 9.

Чтобы смоделировать поток работ, необходимо:

- Установить фокус потока работ. Невозможно показать все важные потоки работ нетривиальной системы в пределах одной диаграммы.
- Выбрать бизнес-объекты, имеющие обязанности самого высокого уровня в отношении всего потока работ или его частей. Это могут быть реальные сущности из словаря системы либо какие-то более абстрактные. В любом случае необходимо создать «плавательную дорожку» для каждого важного бизнес-объекта или подразделения.
- Идентифицировать предусловия начального состояния рабочего потока и постусловия его конечного состояния. Это весьма существенно для очерчивания границ потока работ.
- Начиная со стартового состояния потока работ специфицировать деятельности, разворачивающиеся во времени, и отобразить их на диаграмме.
- Сложные действия или их наборы, которые применяются многократно, представить в виде вызовов отдельных диаграмм деятельности.
- Изобразить потоки, которые соединяются с этими действиями и узлами деятельности: начать с последовательных потоков, затем рассмотреть ветвление и только в последнюю очередь – разделение и соединение.
- Если существуют важные значения объектов, вовлеченных в рабочий поток, отобразить их на диаграмме. Показать их изменяющиеся значения и состояния, насколько это необходимо для передачи назначения потоков объектов.

На рис. 20.10 в качестве примера показана диаграмма деятельности для системы розничной торговли, которая специфицирует поток работ, возникающий при возврате заказчиком продукта, высланного по почте. Работа начинается с действия `Request Return` (Возврат заказа), выполняемого объектом `Customer` (Заказчик), затем поток проходит через отдел `Telesales` (Почтовые продажи) – `Get return number` (Получить номер возврата), обратно к заказчику – `Ship item` (Отправить товар), затем попадает на `Warehouse` (Склад) – `Receive item` (Принять позицию) и `Restock item` (Пополнить запас) – и, наконец, завершается в отделе `Accounting` (Бухгалтерия) – `Credit account` (Кредитовать счет). Как показано на диаграмме, один важный объект – экземпляр `Item` (Позиция товара) – также путешествует в потоке, изменяя свое состояние от `returned` (возвращен) до `available` (доступен).

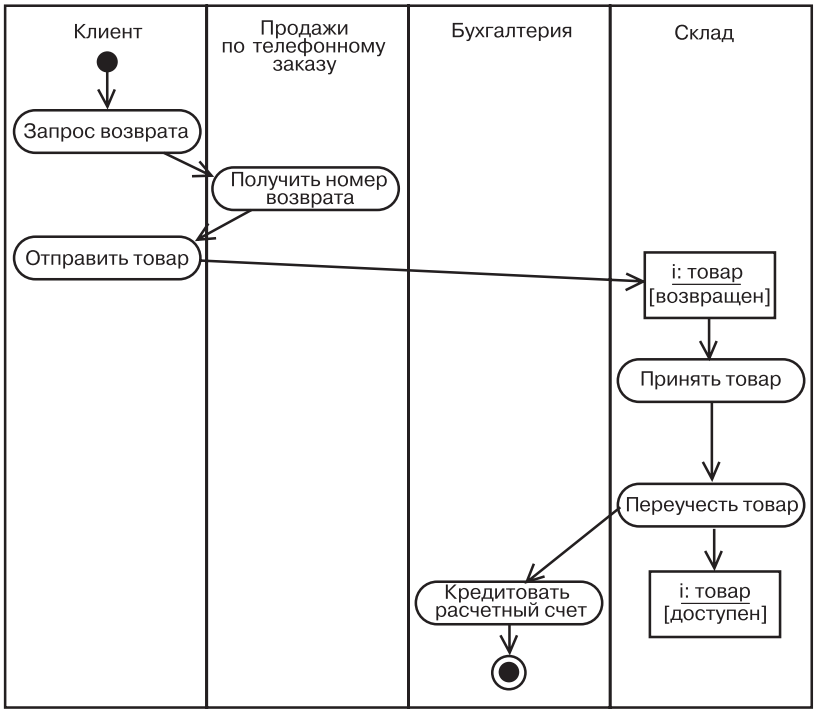


Рис. 20.10. Моделирование потока работ

На заметку. Потоки работ являются бизнес-процессами в большинстве случаев, но не всегда. Например, вы можете использовать диаграмму деятельности для описания процесса разработки программного обеспечения, такого как процесс управления конфигурацией. Более того, диаграммы деятельности подходят для моделирования непрограммных систем, в частности потока пациентов в системе здравоохранения.

В этом примере нет ветвлений, разделений и соединений. Но все это можно встретить в более сложных потоках работ.

Моделирование операций

Диаграммы деятельности могут быть связаны с любым моделируемым элементом для визуализации, специфицирования, конструирования и документирования его поведения. Вы можете подключить такие диаграммы к классам, интерфейсам, компонентам, узлам, вариантам использования и кооперациям. Чаще всего элементом, для которого вы будете разрабатывать диаграмму деятельности, будет операция.

Классы и операции обсуждаются в главах 4 и 9, интерфейсы – в главе 11, компоненты – в главе 15, узлы – в главе 27, варианты использование – в главе 17, кооперации – в главе 28, предусловия, постусловия и инварианты – в главе 9, активные классы – в главе 23.

Использованная подобным образом, диаграмма выступает просто в качестве блок-схемы действий, выполняемых данной операцией. Главное преимущество диаграммы деятельности в том, что все элементы, указанные на ней, семантически связываются в одну модель. Например, любая другая операция или сигнал, на которую/который ссылается действие, может подвергнуться проверке на соответствие классу целевого объекта.

Чтобы смоделировать операцию, необходимо:

- Собрать вместе все абстракции, участвующие в ней. Имеются в виду ее параметры (и тип возвращаемого значения, если таковое есть), атрибуты включающего класса и некоторые соседние классы.
- Идентифицировать предусловия начального состояния операции и постусловия ее конечного состояния, а также любые инварианты включающего класса, которые должны сохраняться на протяжении ее выполнения.
- С начала операции специфицировать все деятельности и действия, разворачивающиеся во времени, и изобразить их на диаграмме вместе с состояниями деятельности и состояниями действий.
- При необходимости использовать ветвление для описания условных путей и итераций.
- В случае если операцией владеет активный класс, использовать разделение и соединение для показа параллельных потоков управления.

Если операция предполагает взаимодействие сообщества объектов, можно смоделировать ее реализацию с помощью кооперации (см. главу 28).

На рис. 20.11 вы видите диаграмму деятельности для класса `Line` (Линия), специфицирующую алгоритм операции `intersection` (пересечение), сигнатура которой включает один параметр (`line` класса `Line`) и одно возвращаемое значение (класса `Point` – Точка). Класс `Line` имеет два интересующих нас атрибута: `slope` (наклон) и `delta` (смещение линии относительно начала координат).

Алгоритм операции прост. Во-первых, есть защитное условие, которое проверяет, не равен ли наклон `slope` данной линии наклону линии-параметра `line`. Если это так, линии не пересекаются и возвращается точка `Point(0,0)`. В противном случае операция сначала вычисляет значение `x` точки пересечения, затем – значение `y`. Как `x`, так и `y` – объекты, локальные по отношению к операции. И наконец, возвращается точка `Point(x,y)`.

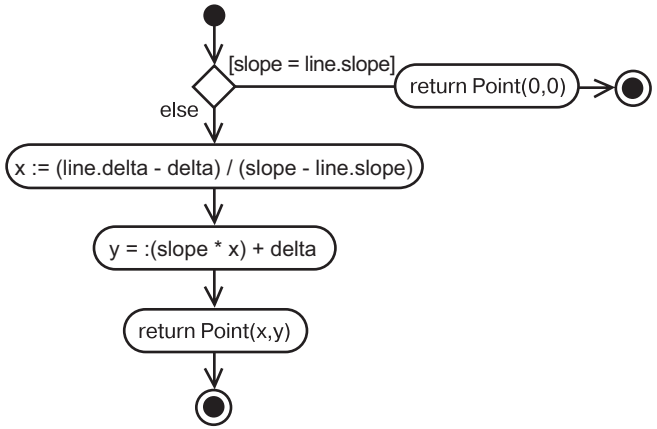


Рис. 20.11. Моделирование операции

На заметку. Использование диаграмм деятельности для изображения блок-схемы операции – отнюдь не самое важное в программировании с помощью UML. Вы *можете* схематично выразить каждую операцию, но практически не должны этого делать. Обычно более целесообразно написание тела операции на определенном языке программирования. Есть смысл прибегать к диаграммам деятельности для моделирования операций, поведение которых трудно для понимания на основе одного лишь исходного кода. Взгляд на блок-схему проясняет алгоритм, который сложно разглядеть в самом коде.

Прямое и обратное проектирование

Прямое проектирование (создание кода из модели) возможно для диаграммы деятельности, особенно если ее контекстом является операция. Например, используя диаграмму, представленную на рис. 20.11, инструментальное средство может сгенерировать следующий код операции `intersection` на C++:

```
Point Line::intersection (Line : line) {
  if (slope == line.slope) return Point(0,0);
  int x = (line.delta - delta) /
                  (slope - line.slope);
  int y = (slope * x) + delta;
  return Point(x, y);
}
```

Здесь применена некоторая хитрость, связанная с инициализацией локальных переменных. Менее «интеллектуальный» инструмент, вероятно, сначала объявил бы переменные, а потом присвоил им значения.

Обратное проектирование (создание модели из кода) также возможно для диаграмм деятельности, особенно если контекст кода – тело операции. Диаграмма, которую мы только что упоминали, могла быть сгенерирована на основе реализации класса `Line`.

Более интересной по сравнению с обратным проектированием модели из кода может оказаться анимация модели по работающей системе. Скажем, имея в распоряжении все ту же диаграмму (рис. 20.11), инструмент мог бы анимировать состояния действий на ней по мере их выполнения в работающей системе. Еще лучше было бы с этим инструментом, находящимся под управлением отладчика, контролировать скорость выполнения, возможно, устанавливая точки прерывания для приостановки процесса в нужные моменты, с тем чтобы проверить значение атрибутов конкретных объектов.

Советы и подсказки

Когда вы создаете диаграммы деятельности в UML, помните, что это лишь проекции одной и той же модели динамических аспектов системы. Ни одна отдельная диаграмма деятельности не может охватить все эти аспекты в совокупности. Поэтому для моделирования динамических аспектов потока работ или операции понадобится использовать множество диаграмм.

Хорошо структурированная диаграмма деятельности должна обладать следующими свойствами:

- фокусироваться на выражении лишь одного аспекта динамики системы;
- содержать только те элементы, которые важны для понимания данного аспекта;
- представлять уровень детализации, согласованный с уровнем абстракции: показывать только те дополнения, которые важны для понимания предмета;
- не быть настолько лаконичной, чтобы читатель упустил из виду важную семантику.

Когда создается диаграмма деятельности, необходимо:

- присвоить ей имя, соответствующее ее назначению;
- начинать с моделирования главного потока. Применять ветвление, параллельность и поток объектов во вторую очередь – возможно, на отдельных диаграммах;
- использовать примечания и цветовые выделения, чтобы привлечь внимание к важным моментам.

Часть V

Расширенное моделирование поведения

Глава 21. События и сигналы

В этой главе:

- **События сигнала, вызова, времени и изменения**
- **Моделирование серии сигналов**
- **Моделирование исключений**
- **Обработка событий в активных и пассивных объектах**

Реальный мир ежедневно преподносит нам новые явления. Мало того, события в нашей жизни порой происходят одновременно и неожиданно. Под событием подразумевается некий значимый факт, локализованный во времени и пространстве.

В контексте конечных автоматов события используются для моделирования определенного воздействия, которое может вызвать переход из одного состояния в другое. К числу событий относятся сигналы, вызовы, истечение определенного промежутка времени или изменение состояния.

События могут быть синхронными и асинхронными; их моделирование – одна из составляющих моделирования процессов и потоков.

Введение

Статичная система не представляет ни малейшего интереса, поскольку в ней ничего не происходит. Все системы, имеющие хоть что-то общее с реальностью, в той или иной мере динамичны, причем динамику обуславливают именно события, происходящие внутри системы или за ее пределами. Работу банкомата инициирует пользователь, который нажимает кнопку для совершения банковской транзакции. Автономный робот начинает действовать, когда встречается с некоторым предметом. Сетевой маршрутизатор реагирует на обнаружение переполнения буферов сообщений. На химическом заводе сигналом к действию становится окончание периода, необходимого для завершения реакции.

В UML любое явление, которое может иметь место в действительности, моделируется как событие. *Событие* (event) – это

описание существенного факта, состоявшегося в определенном времени и пространстве. Получение сигнала, истечение промежутка времени, изменение состояния – это примеры асинхронных событий, которые могут произойти в любой момент. Вызовы – это, как правило, синхронные события, используемые для запуска некоей операции.

Графическое представление событий в UML продемонстрировано на рис. 21.1. Такая нотация позволяет визуализировать объявления событий – например, сигнал `OffHook` (ТрубкаПовешена) и показать, как наступление события приводит к переходу между состояниями: например, сигнал `OffHook` вызывает переход телефона из состояния `Active` (Активен) в состояние `Idle` (Ожидание) и выполнение действия `dropConnection` (разорватьСоединение).

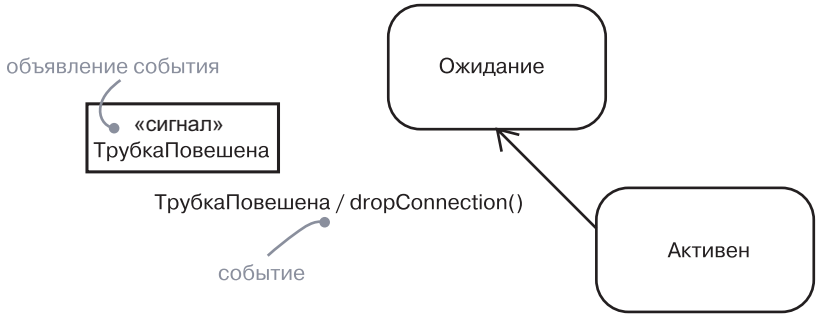


Рис. 21.1. События

Базовые понятия

Событие – это описание существенного факта, локализованного во времени и пространстве. Применительно к автоматам событие означает воздействие, которое может вызвать переход из одного состояния в другое.

Сигнал – это разновидность события, при использовании которой сообщение передается асинхронно от одного экземпляра к другому.

Виды событий

Действующие лица обсуждаются в главе 17, системы – в главе 32.

События могут быть внутренними или внешними. *Внешние события* передаются между системой и ее действующими лицами (примерами могут служить нажатие кнопки или прерывание от датчика предотвращения столкновений), а *внутренние* – между объектами, существующими в самой системе (пример – исключение, генерируемое при переполнении).

Создание и уничтожение объектов – это тоже разновидность сигналов (см. главу 16). Классы обсуждаются в главах 4 и 9, обобщения – в главах 5 и 10.

В UML можно моделировать четыре вида событий: сигналы, вызовы, истечение промежутка времени и изменение состояния.

Сигналы

Сообщение – это именованный объект, который асинхронно посылается одним объектом и принимается другим. *Сигнал* (signal) представляет собой классификатор для сообщений и сам является типом сообщения.

У сигналов есть много общего с простыми классами. Например, можно говорить об экземплярах сигналов, хотя обычно не возникает необходимость моделировать их явно. Кроме того, сигналы могут участвовать в связях обобщения, что позволяет моделировать иерархии событий, где одни – например, сигнал `NetworkFailure` (СбойСети) – являются общими, а другие – например, специализация события `NetworkFailure` под названием `WarehouseServerFailure` (`ОтказСкладскогоСервера`) – частными. Как и классы, сигналы могут иметь атрибуты и операции.

На заметку. Атрибуты сигнала выступают в роли его параметров. Например, при передаче сигнала `Столкновение` можно указать значения его атрибутов в виде параметров: `Столкновение(3, 5)`.

Конечные автоматы обсуждаются в главе 22, взаимодействие – в главе 16, интерфейсы – в главе 11, зависимости – в главе 5, стереотипы – в главе 6.

Сигнал может отправляться в результате выполнения некоего действия в процессе перехода (изменения состояния) в автомате. Его можно смоделировать как сообщение, передаваемое между двумя ролями при некотором их взаимодействии. При выполнении метода тоже могут передаваться сигналы. Когда моделируется класс или интерфейс, важная часть спецификации поведения – указание сигналов, которые могут посылать его операции.

Как показано на рис. 21.2, сигналы в UML моделируются классами со стереотипами. Для указания на то, что некоторая операция посылает сигнал, можно воспользоваться зависимостью со стереотипом `send`.

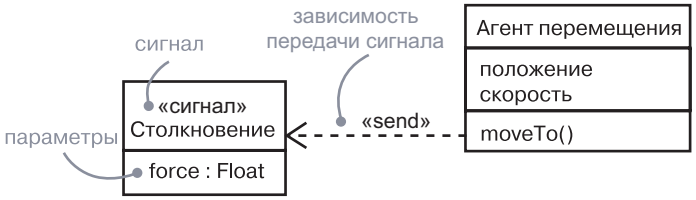


Рис. 21.2. Сигналы

События вызова

Конечные автоматы обсуждаются в главе 22.

Если событие сигнала представляет собой его экземпляр, то под *событием вызова* (call event) понимается получение некоторым объектом запроса на выполнение операции над ним. Событие вызова может привести к переходу между состояниями в автомате или вызову метода на целевом объекте. Конкретный вариант задается в определении операции класса.

В то время как сигнал является событием асинхронным, событие вызова обычно синхронно. Это означает, что, когда один объект инициирует выполнение операции на другом объекте, у которого есть свой автомат, управление передается от отправителя получателю, срабатывает соответствующий переход, затем операция завершается, получатель переходит в новое состояние и возвращает управление отправителю. В тех случаях, когда отправитель не нуждается в ожидании ответа, вызов может быть определен как асинхронный.

По рис. 21.3 видно, что в модели событие вызова неотличимо от события сигнала. В обоих случаях событие вместе со своими параметрами выглядит как триггер для перехода состояния.

Рис. 21.3. События вызова

На заметку. Хотя визуально событие сигнала и вызова неотличимы, разница совершенно отчетливо проявляется на заднем плане модели. Получатель события, конечно же, знает о различии (путем объявления операций в своем списке операций). Сигнал, как правило, обрабатывается на уровне автомата, а событие вызова – методом. Для перехода от события к сигналу или операции можно воспользоваться инструментальными средствами.

События времени и изменения

Событие времени представляет собой истечение некоего промежутка времени. На рис. 21.4 показано, что в UML такое событие моделируется посредством ключевого слова `after` (после), за которым следует выражение, определяющее этот промежуток. Выражение может быть простым – например, `after 2 seconds` (через 2 секунды)

или сложным – например, `after 1 ms since exiting Idle` (через 1 мс после выхода из состояния Ожидание). Если явно не указано иное, отсчет времени начинается с момента входа в текущее состояние. Для обозначения конца периода используется слово `at`. Так, запись `at (1 Jan 2007, 1200 UT)` указывает на то, что событие времени закончится в первый день нового 2007-го года.

С помощью *события изменения* описывается изменение состояния или выполнения некоторого условия. На рис. 21.4 показано, что в UML событие изменения моделируется посредством ключевого слова `when` (когда), за которым следует булево выражение. Такое выражение может использоваться для обозначения абсолютного момента времени (например, `when time=11:59`, «когда время=11:59») или проверки условия (например, `when altitude<1000`, «когда высота<1000»).

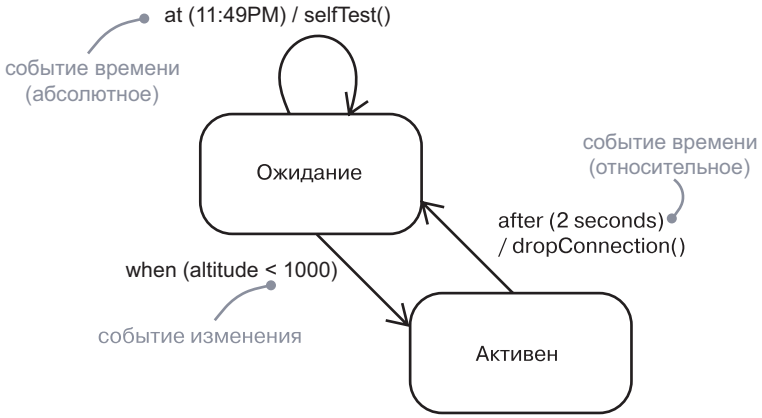


Рис. 21.4. События времени и изменения

Событие изменения происходит один раз при изменении значения условия с ложного на истинное (но не наоборот). Пока условие остается истинным, событие не повторяется.

На заметку. Хотя событие изменения моделирует условие, проверяемое непрерывно, обычно, проанализировав ситуацию, можно выявить дискретные моменты времени, когда это условие нужно проверять.

Передача и получение событий

Процессы и потоки обсуждаются в главе 23.

В событиях сигнала и вызова участвуют по меньшей мере два объекта: тот, который посылает сигнал или инициирует операцию, и тот, которому событие адресовано. Поскольку сигналы по своей природе асинхронны, а асинхронные вызовы сами по себе являются

сигналами, семантика событий перекликается с семантикой активных и пассивных объектов.

Экземпляры обсуждаются в главе 13.

Экземпляр любого класса может послать сигнал принимающему объекту или вызвать его операцию. Отправив сигнал получателю, он продолжает свой поток управления, не дожидаясь от него ответа. Например, после того как действующее лицо, взаимодействующее с банкоматом, пошлет сигнал `pushButton` (нажатьКнопку), оно может выполнять другие действия независимо от того, что делает система, которой был послан сигнал. Напротив, если объект вызывает операцию, он должен дождаться ответа от получателя. Допустим, в трейдерской системе экземпляр класса `Trader` (Трейдер) может вызвать операцию `confirmTransaction` (подтвердитьТранзакцию) в некотором экземпляре класса `Trade` (Сделка), тем самым косвенно изменив состояние последнего. Если это синхронный вызов, то объект `Trader` будет ждать, пока операция закончится.

На заметку. В некоторых случаях возникает необходимость в изображении объекта, который посылает сигнал сразу нескольким объектам (мультивещание, multicasting) или всем ожидающим объектам в системе (широковещание, broadcasting). Для моделирования мультивещания следует изобразить объект, посылающий сигнал набору, в который входят все получатели. Для моделирования широковещания нужно показать, что один объект посылает сигнал другому, представляющему систему в целом.

Конечные автоматы обсуждаются в главе 22, активные объекты – в главе 23.

Любой экземпляр любого класса может быть получателем события вызова или сигнала. Если это синхронное событие вызова, то отправитель и получатель находятся в состоянии «рандеву» на всем протяжении выполнения операции. Это означает, что поток управления отправителя блокируется, пока операция не завершится. Если это сигнал, то отправитель и получатель не входят в состояние «рандеву»: отправитель посылает сигнал, но не дожидается ответа от получателя. В любом случае событие может быть потеряно (если явно не указано, что нужен ответ), вызвать переход состояния в автомате получателя, если он существует, или инициировать обычный вызов метода.

На заметку. Вызов может быть асинхронным. В этом случае отправитель продолжает выполнение своих действий сразу после отправления сигнала. Передача сообщения и его обработка получателем происходят параллельно с действиями отправителя. Когда выполнение метода завершается, он просто заканчивается. Если метод пытается вернуть значения, то они игнорируются.

Операции и классы обсуждаются в главе 4.

В UML события вызова, которые получает объект, моделируются как операции класса этого объекта. Именованные сигналы, получаемые объектом, моделируются путем перечисления в дополнительном разделе класса, как показано на рис. 21.5.

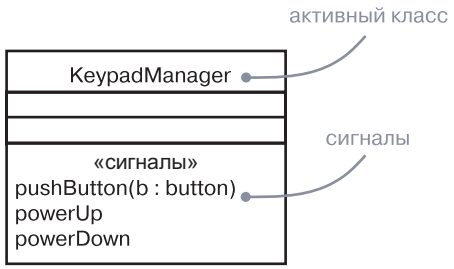


Рис. 21.5. Сигналы и активные классы

Интерфейсы обсуждаются в главе 11, асинхронные операции – в главе 23.

На заметку. Точно так же можно присоединить именованные сигналы к интерфейсу. В любом случае имена сигналов, перечисленные в дополнительном разделе, являются не объявлениями, а лишь указаниями на то, что сигнал используется.

Типичные приемы моделирования

Моделирование семейства сигналов

Обобщения обсуждаются в главе 5 и 10.

В большинстве систем, управляемых событиями, события сигналов образуют иерархию. Например, автономный робот может различать внешние сигналы, такие как `Collision` (Столкновение), и внутренние, например `HardwareFault` (АппаратныйОтказ). Однако множества внешних и внутренних сигналов могут пересекаться. И даже внутри этих двух широких областей классификации можно обнаружить частные разновидности. К примеру, `HardwareFault` можно подразделить на `BatteryFault` (ОтказБатареи) и `MovementFault` (ОтказДвигательногоМеханизма). Допускается и дальнейшая специализация. Так, разновидностью сигнала `MovementFault` является `MotorStall` (ОстановЭлектромотора).

Конечные автоматы обсуждаются в главе 22

Моделируя подобным образом иерархии сигналов, можно специфицировать полиморфные события. Рассмотрим, к примеру, автомат, в котором некоторый переход срабатывает только при получении сигнала `MotorStall`. Поскольку этот сигнал является в иерархии листовым, то переход инициируется только им, так что полиморфизм отсутствует. Но предположим теперь, что вы построили автомат,

в котором переход срабатывает при получении сигнала `HardwareFault`. В этом случае переход полиморфен: его вызывает как сам сигнал `HardwareFault`, так и любая его специализация (разновидность), включая `BatteryFault`, `MovementFault` и `MotorStall`.

Для моделирования семейства сигналов следует:

- Рассмотреть все разновидности сигналов, на которые может отвечать данное множество активных объектов.
- Выявить схожие виды сигналов и поместить их в иерархию типа «обобщение/специализация», используя наследование. На верхних уровнях иерархии будут располагаться общие сигналы, а на нижних – специализированные.
- Определить возможности полиморфизма в автоматах этих активных объектов. При обнаружении полиморфизма скорректировать иерархию, добавив в случае необходимости промежуточные абстрактные сигналы.

Абстрактные классы обсуждаются в главе 5 и 9.

На рис. 21.6 приведена модель семейства сигналов, которые могли бы обрабатываться автономным роботом. Обратите внимание, что корневой сигнал `RobotSignal` (СигналРоботу) является абстрактным, то есть непосредственное создание его экземпляров невозможно. У этого сигнала есть две конкретные специализации (`Collision` и `HardwareFault`), одна из которых (`HardwareFault`) специализируется и дальше. Заметьте, что у сигнала `Collision` есть один параметр.

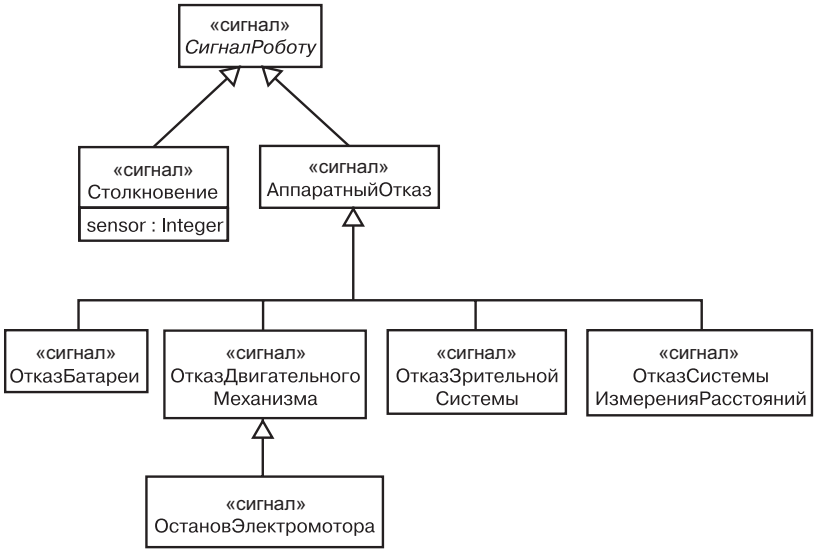


Рис. 21.6. Моделирование семейства сигналов

Моделирование аварийных ситуаций

Классы обсуждаются в главе 4 и 9, интерфейсы – в главе 11, стереотипы – в главе 6.

Важная часть визуализации, специфицирования и документирования поведения класса или интерфейса – выявление аварийных ситуаций, которые могут порождать его операции. Если вы имеете дело с некоторым классом или интерфейсом, то его операции ясны из описания, но понять, какие аварийные ситуации они могут вызвать, нелегко, если это явно не указано в модели.

В UML аварийные ситуации представляют собой разновидность событий и моделируются как сигналы. События-ошибки можно присоединить к спецификации операций. Моделирование исключений в некотором смысле противоположно моделированию семейств сигналов. Основная цель последнего – специфицировать, какие сигналы могут получать активные объекты; цель же моделирования аварийных ситуаций состоит в том, чтобы показать, какие аварийные ситуации может порождать объект.

Чтобы выполнить эту задачу, требуется:

- Для каждого класса и интерфейса и для каждой определенной в них операции рассмотреть нормальные и аварийные ситуации и смоделировать их в виде сигналов, передаваемых между объектами.
- Организовать иерархию сигналов: на верхних уровнях разместить общие сигналы, на нижних – специализированные, и при необходимости ввести промежуточные исключения.
- Указать для каждой операции, какие аварийные ситуации (сигналы) она может порождать. Это можно сделать явно на диаграмме (проведя зависимости со стереотипом `send` от операции к ее сигналам) или же использовать диаграммы последовательности, изображающие различные сценарии.

Классы-шаблоны обсуждаются в главе 9.

На рис. 21.7 представлена модель иерархии аварийных ситуаций, которые могут порождаться контейнерными классами из стандартной библиотеки, например классом-шаблоном `Set` (Установка). В корне этой иерархии находится абстрактный сигнал `SetError` (УстановитьОшибку), а ниже расположены специализированные виды ошибок: `Duplicate` (Дублирование), `Overflow` (Переполнение) и `Underflow` (Потеря значимости). Как видно, операция `add` (добавить) может породить сигналы `Duplicate` и `Overflow`, а операция `remove` (удалить) – только сигнал `Underflow`. Вместо этого можно было бы убрать зависимости с переднего плана, перечислив их в спецификации каждой операции. Как бы то ни было, зная, какие сигналы может породить операция, вы сможете успешно создавать программы, использующие класс `Set`.

На заметку. Сигналы, в том числе аварийных ситуаций, – это асинхронные события, происходящие между объектами. UML также предусматривает исключения, которые можно обнаружить в таких языках программирования, как Ada или C++. Исключение является условием, при котором основной поток выполнения задачи прерывается и выполняется другой поток. Исключения – это не сигналы; они просто представляют собой удобный механизм для спецификации альтернативного потока управления внутри единственного синхронного потока выполнения задачи.

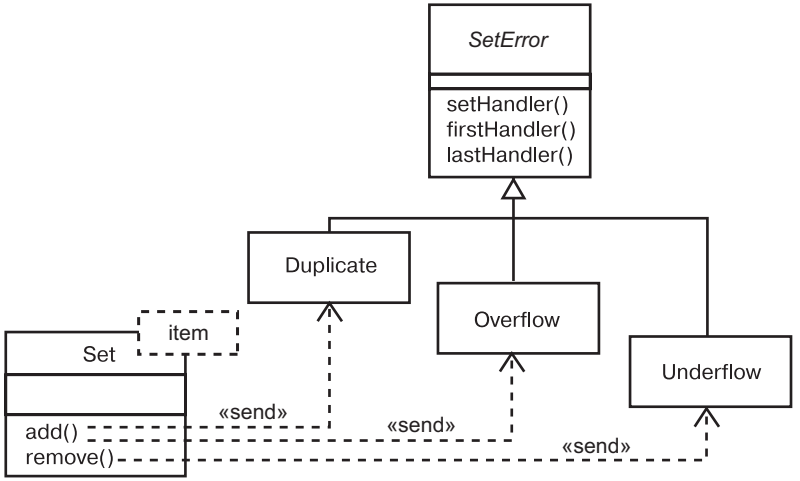


Рис. 21.7. Моделирование условий ошибки

Советы и подсказки

При моделировании событий исходите из следующих соображений:

- стройте иерархию сигналов так, чтобы можно было воспользоваться общими для них свойствами;
- не забывайте ассоциировать подходящий автомат с каждым элементом, который может получать события;
- обязательно моделируйте не только те элементы, которые могут получать события, но и те, которые могут их посылать.

Изображая событие в UML, в общем случае моделируйте иерархии событий явно, а их использование специфицируйте на заднем плане тех классов или операций, которые посылают или получают событие.

Глава 22. Конечные автоматы

В этой главе:

- Состояния, переходы и деятельности
- Моделирование жизненного цикла объекта
- Создание хорошо структурированных алгоритмов

Взаимодействия обсуждаются в главе 16, объекты – в главе 13.

Используя взаимодействие, можно моделировать поведение сообщества совместно работающих объектов. При помощи конечного автомата можно смоделировать поведение отдельного объекта. Автомат – это поведение, которое специфицирует последовательность состояний объекта, через которые он проходит в течение своего жизненного цикла в ответ на события, а также реакции на эти события.

Классы обсуждаются в главах 4 и 8, варианты использования – в главе 17, системы в главе 32, диаграммы деятельности – в главе 20, диаграммы состояний – в главе 25.

Автоматы используются для моделирования динамических аспектов поведения системы. Большей частью этот процесс подразумевает спецификацию жизненного цикла экземпляров класса, варианта использования или системы в целом. Эти экземпляры могут реагировать на такие события, как сигналы, операции либо истечение некоего периода времени. Когда происходит событие, в зависимости от текущего состояния объекта наблюдается некоторый эффект. *Эффект* – это спецификация поведения, реализуемого внутри автомата. В конечном счете эффекты проявляют себя в выполнении действий, изменяющих состояние объекта или возвращающих какое-либо значение. *Состояние* объекта – это период времени, в течение которого он удовлетворяет заданным условиям, выполняет некую деятельность или ожидает определенного события.

Вы можете визуализировать динамику выполнения некоторого процесса двумя способами: подчеркивая поток управления от одной деятельности к другой (при помощи диаграммы деятельности) или же выделяя потенциальные состояния объекта и переходы между ними (при помощи диаграммы состояний).

Хорошо структурированные автоматы подобны хорошо структурированным алгоритмам: они эффективны, просты, адаптируемы и понятны.

Введение

Рассмотрим жизненный цикл домашнего термостата в погожий осенний день. Ранним утром ничто не тревожит его покой. Температура в доме стабильна, а если не налетит ураган и не поднимется сильный ветер, то и температура снаружи не должна меняться. Ближе к рассвету наблюдается иная картина. Солнце встает из-за горизонта, и температура воздуха на улице слегка повышается. Начинают просыпаться члены семьи; вот кто-то вылез из кровати и подкрутил регулятор. Оба этих события важны для системы терморегуляции дома. Термостат начинает вести себя, как все порядочные устройства его класса: отдает команду либо обогревателю (чтобы подогреть воздух), либо кондиционеру (чтобы его охладить).

После того как обитатели дома ушли на работу или в школу, движение прекращается, и температура вновь становится постоянной. Но тут может вмешаться автоматика, которая приказывает термостату понизить температуру, чтобы не расходовать зря электроэнергию и газ. Термостат снова принимается за работу. Ближе к концу дня автоматическая программа снова «оживает», но на этот раз велит поднять температуру, потому что скоро семейство окажется в полном сборе, и к этому моменту дом должен стать уютным.

Вечером, когда дом наполнен теплом, исходящим от человеческих тел, а на плите готовится еда, у термостата масса работы: он должен включать то обогреватель, то кондиционер. Наконец, ночью все снова успокаивается.

Многие программные системы ведут себя подобно термостату. Сердечный стимулятор работает круглосуточно, но адаптируется к изменениям кровяного давления или выполняемой человеком работе. Сетевой маршрутизатор тоже работает непрерывно, незаметно перенаправляя потоки битов, а иногда изменяя свое поведение в ответ на команды администратора сети. Сотовый телефон работает по запросам, отвечая на действия своего владельца и на сообщения от локальных ретрансляторов сотовой связи.

Моделирование структурных аспектов системы обсуждается в частях II и III.

В UML моделирование статических аспектов системы выполняется с помощью диаграмм классов и объектов. Они позволяют визуализировать, специфицировать, конструировать и документировать те сущности, которые «живут» внутри системы, включая классы, интерфейсы, компоненты, узлы, варианты использования и их экземпляры, а также связи между ними.

Динамические аспекты системы в UML моделируются *конечными автоматами* (state machines). В то время как взаимодействие моделирует сообщество объектов, совместно работающих для выполнения некоторых действий, автомат моделирует жизненный цикл отдельного объекта – будь то экземпляр класса, варианта использования

Динамические аспекты систем также можно моделировать при помощи взаимодействий (см. главу 16); события обсуждаются в главе 21. Диаграммы деятельности обсуждаются в главе 20, диаграммы состояний – в главе 25.

или даже всей системы. За время жизненного цикла в объекте может происходить множество разнообразных событий, таких как передача и прием сигналов, вызовы операций, создание и уничтожение объекта, истечение периодов времени, отведенного на некое действие, либо изменение каких-то условий. В ответ на эти события объект выполняет определенное действие, представляющее собой вычисление, а затем изменяет свое состояние на другое. Поэтому поведение такого объекта зависит от прошлого, по крайней мере в той степени, в которой оно влияет на его текущее состояние. Объект может принять событие, ответить на него действием, затем изменить свое состояние. Когда же он принимает другое событие, его реакция может быть другой – в зависимости от текущего состояния, которое является результатом предыдущего события.

Автоматы используются для моделирования поведения любого элемента – чаще всего класса, варианта использования или всей системы. Автоматы могут быть визуализированы на диаграммах состояний. Вы можете сосредоточиться на поведении объекта, зависящем от последовательности событий, что особенно удобно для моделирования реактивных систем.

Графическое представление состояний, переходов, событий и эффектов в UML продемонстрировано на рис. 22.1. Эта нотация позволяет показать поведение объекта таким образом, чтобы выделить важные элементы его жизненного цикла.

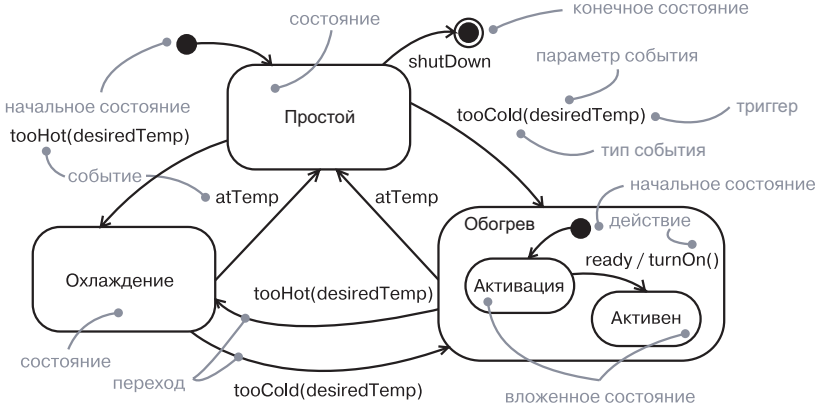


Рис. 22.1. Автоматы

Термины и понятия

Автомат – это поведение, специфицирующее последовательность состояний, через которые проходит объект за время своего существования в ответ на события, а также его реакцию на эти события.

Состояние – ситуация во время жизни объекта, в которой он удовлетворяет заданным условиям, осуществляет некую деятельность либо пребывает в ожидании событий.

Событие – это спецификация значимого происшествия, локализованного во времени и пространстве. Применительно к автоматам событие – это воздействие, которое может инициировать переход от одного состояния к другому.

Переход – связь между двумя состояниями, указывающая на то, что объект в первом состоянии выполнит определенные действия и перейдет во второе, когда случится определенное событие и заданное условие будет удовлетворено. Изображается сплошной линией со стрелкой или путем, направленным от исходного состояния к новому.

Деятельность (activity) – происходящий в данный момент неатомарный процесс внутри автомата.

Действие (action) – исполняемое вычисление, в результате которого изменяется состояние модели или возвращается некоторое значение. Изображается в виде прямоугольника с закругленными углами.

Контекст

Объекты обсуждаются в главе 13, сообщения – в главе 16.

Каждый объект имеет свой жизненный цикл: сначала создается, затем прекращает свое существование. Между этими двумя событиями объект может влиять на другие объекты, посылая им сообщения, а также подвергаться их влиянию, принимая сообщения от них. Во многих случаях такие сообщения представляют собой простые синхронные вызовы операций. Например, экземпляр класса `Customer` (Покупатель) может вызвать операцию `getAccountBalance` (ЗапроситьБаланс) на экземпляре класса `BankAccount` (БанковскийСчет). Подобным объектам не нужен автомат для описания их поведения, поскольку их текущее поведение не зависит от прошлого.

Сигналы обсуждаются в главе 21.

В системах других типов можно встретить объекты, которые должны отвечать на сигналы, представляющие собой асинхронные сообщения, передаваемые между экземплярами. Например, сотовый телефон должен отвечать на случайные телефонные вызовы с других телефонов, на события нажатия клавиш пользователем, а также на сигналы из сети, когда телефон перемещается из одной «соты» в другую. Кроме того, можно встретить объекты, текущее поведение которых зависит от прошлого. Например, поведение системы управляемой ракеты «воздух-воздух» зависит от ее текущего состояния, такого как `NotFlying` (НеЛетит) – не слишком хорошая идея запускать ракету с самолета, стоящего на земле! – или `Searching` (Поиск): не стоит пускать ракету, пока не выбрана цель.

Активные объекты обсуждаются в главе 23, моделирование реактивных систем – в главе 25, варианты использования – в главе 17, взаимодействия – в главе 16, интерфейсы – в главе 11.

Поведение объекта, который должен отвечать на асинхронные сообщения либо его текущее поведение зависит от прошлого, лучше всего специфицировать при помощи автоматов. Это касается экземпляров классов, которые могут принимать сигналы, включая многие активные объекты. Фактически объект, который принимает сигнал, но не имеет для него перехода в своем текущем состоянии и не откладывает реакцию на сигнал в этом состоянии, будет просто его игнорировать. Другими словами, отсутствие перехода для сигнала не является ошибкой; это значит, что в данной точке сигнал не представляет интереса. Вы также будете использовать автоматы для моделирования поведения систем в целом, особенно реактивных, которые должны отвечать на сигналы действующих лиц за пределами системы.

На заметку. В основном для моделирования поведения вариантов использования применяются взаимодействия, но с той же целью можно использовать и автоматы. Также они подходят для моделирования поведения интерфейсов. Хотя интерфейс может и не иметь конкретных экземпляров, класс, реализующий его, – может. Такой класс должен соответствовать поведению, специфицированному автоматом для интерфейса.

Состояния

Состояние объекта можно визуализировать во взаимодействии, как описано в главе 13, последние четыре части состояния обсуждаются ниже в этой главе.

Состояние (state) – это ситуация в жизненном цикле объекта, в которой он удовлетворяет заданным условиям, осуществляет некую деятельность или ожидает определенного события. Объект пребывает в том или ином состоянии ограниченный период времени. Например, обогреватель в доме может быть в любом из четырех состояний: простой (ожидание команды на включение обогрева), активация (газ подан, но ожидается достижение определенной температуры), функционирование (газ и котел включены), выключение (подача газа прекратилась, но котел включен, остаточное тепло сбрасывается из системы).

Когда автомат объекта находится в определенном состоянии, говорят, что в этом состоянии находится сам объект. Например, экземпляр `Heater` (Обогреватель) может быть в состоянии `Idle` (Простой) или `ShuttingDown` (Выключение).

Состояние характеризуется несколькими частями:

- *имя* – текстовая строка, отличающая данное состояние от других. Состояние может быть анонимным, то есть без имени;
- *входной/выходной эффекты* – действия, выполняемые соответственно при входе и выходе из состояния;

- *внутренние переходы* – переходы, которые обрабатываются без смены состояния;
- *подсостояния* – вложенные состояния, в том числе неортогональные (последовательно активные) или ортогональные (параллельно активные) подсостояния;
- *отложенные события* – список событий, которые не обрабатываются в данном состоянии, но откладываются и помещаются в очередь для обработки объектом в другом состоянии.

На заметку. Имя состояния – это текст, который может состоять из букв и цифр в неограниченном количестве, а также некоторых знаков препинания (за исключением таких, как двоеточие), и записываться в несколько строк. На практике имена состояний представляют собой краткие существительные, взятые из словаря моделируемой системы. Обычно первая буква каждого слова в имени состояния – заглавная, например `Idle` или `ShuttingDown`.

Как показано на рис. 22.2, состояние изображается в виде прямоугольника с закругленными углами.

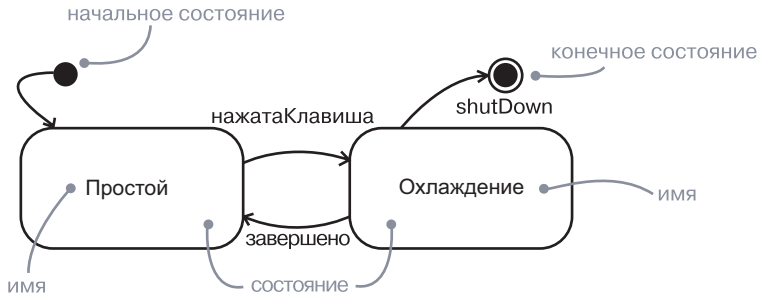


Рис. 22.2. Состояния

Начальное и конечное состояния. Как показано на рисунке, существует два особых состояния, которые могут быть определены для автомата объекта. Первое из них – начальное состояние, которое означает начальную точку, по умолчанию установленную для автомата или подсостояния. Оно изображается закрашенным черным кружком. Второе состояние – конечное; оно означает, что работа автомата или включающее состояние завершены. Конечное состояние представлено в форме закрашенного черного кружка, вписанного в окружность.

На заметку. На самом деле начальное и конечное состояния – это псевдосостояния. Ни одно из них не наделено признаками «нормальных» состояний, за исключением имени. А вот переход от начального состояния к обычному может обладать всеми характерными для переходов свойствами, включая защитное условие и действие (но не инициирующее событие).

Переходы

Переход – это связь между двумя состояниями, означающая, что объект в первом состоянии должен выполнить определенные действия и перейти во второе состояние, когда произойдет определенное событие и будут удовлетворены заданные условия. До тех пор пока переход не произошел, об объекте говорят, что он находится в *исходном состоянии*; после перехода он пребывает в *целевом состоянии*. Например, `Heater` (Обогреватель) может перейти из состояния `Idle` (Простой) в состояние `Active` (Активация), когда произойдет событие `tooCold` (слишкомХолодно) с параметром `desiredTemp` (нужнаяТемпература).

Переход состоит из пяти частей:

- *исходное состояние* – состояние, на которое воздействует переход. Если объект находится в исходном состоянии, то исходящий переход может произойти, когда объект примет событие, инициирующее переход, и будет выполнено защитное условие (если оно есть);
- *инициирующее событие* – событие, которое, будучи распознанным объектом в исходном состоянии, инициирует выполнение перехода, обеспечивая истинность его защитного условия;
- *защитное условие* – булево выражение, которое вычисляется при инициировании перехода после получения события. Если выражение истинно, то выполнение перехода разрешается. В противном случае переход не выполняется, и если нет других переходов, которые могут быть инициированы тем же событием, то событие теряется;
- *эффект* – исполняемое поведение (такое как действие), которое может быть присуще объекту, владеющему данным автоматом, и опосредованно – с другими объектами, которые видимы данному;
- *целевое состояние* – состояние, в котором оказывается объект по завершении перехода.

Как показано на рис. 22.3, переход изображается в виде сплошной линии со стрелкой, направленной от исходного состояния к целевому. *Переход в себя* (self-transition) – это переход, исходное и целевое состояния которого совпадают.

На заметку. Переход может иметь множество исходных состояний (в этом случае он представляет собой слияние множества параллельных состояний), а также множество целевых состояний (в этом случае он представляет разделение на множество параллельных состояний). Об этом подробнее будет рассказано ниже, при обсуждении ортогональных подсостояний.

Триггер события

События обсуждаются в главе 21.

Событие – это спецификация значимого происшествия, локализованного во времени и пространстве. В контексте автоматов событие – это воздействие, которое может вызвать переход из одного состояния в другое. Как показано на рис. 22.3, под событиями могут подразумеваться поступление сигнала, вызов, истечение определенного периода времени либо изменение состояния. Сигнал или вызов могут иметь параметры, которые доступны переходу, в том числе выражения защитного условия и действия.

Специфицирование семейства сигналов обсуждается в главе 21. Множественные неперекрывающиеся защитные условия формируют ветвление (см. главу 20).

Допускается существование завершающего перехода (то есть такого, который не имеет инициирующего события). Завершающий переход инициируется неявно, когда исходное состояние завершает свою деятельность, если таковая наблюдается.

На заметку. Триггер события может быть полиморфным. Например, если вы специфицировали семейство сигналов, то переход, у которого триггер события – S, может быть вызван S, а также всеми его потомками.

Защитное условие

Как показано на рис. 22.3, защитное условие представлено в виде булева выражение, заключенное в квадратные скобки и помещенное после триггера события. Таким образом, допускается наличие множества переходов из одного исходного состояния, с одним и тем же триггером события, – до тех пор пока условия не перекрываются.

Защитное условие вычисляется лишь однажды для каждого перехода в момент возникновения события, но оно может вычисляться

и повторно, если переход инициируется еще раз. Внутри булева выражения можно задать условия пребывания объекта в том или ином состоянии – например, выражение «Обогреватель в состоянии Простоя» принимает значение «истина», если объект `Heater` (Обогреватель) находится в состоянии `Idle` (Простой). Если условие оказывается не соблюденным при его проверке, то событие не повторяется позднее, когда условие станет истинным. Чтобы смоделировать такое поведение, следует использовать событие изменения.

События изменений обсуждаются в главе 21.

На заметку. Если защитное условие вычисляется только однажды при каждом инициировании перехода, то событие изменения потенциально вычисляется постоянно.

Эффект

Эффект – это поведение, которое имеет место, когда инициируется переход. Под эффектами могут подразумеваться встроенные вычисления, вызовы операций (для объекта, владеющего автоматом, а также для других видимых объектов), создание и уничтожение другого объекта либо передача сигнала некоторому объекту. Чтобы отметить факт отправки сигнала, можно снабдить имя сигнала префиксом – ключевым словом `send`.

Деятельности обсуждаются ниже в настоящей главе, зависимости – в главах 5 и 10.

Переходы допускаются только при условии, что автомат неактивен (находится в состоянии покоя), то есть не выполняется эффект (действие) от предыдущего перехода. Срабатывание эффекта перехода и любых ассоциированных входных и выходных эффектов должно завершиться прежде, чем любым дополнительным событиям будет позволено вызвать дополнительные переходы. В этом состоит отличие от выполнения деятельности (см. ниже), которое может быть прервано некоторыми событиями.

На заметку. Вы вправе явно показать объект, которому послан сигнал, используя зависимость со стереотипом `send`, источником которого является состояние, а целью – объект.

Расширенные состояния и переходы

Моделирование самого разного поведения в UML в большинстве случаев требует использования только базовых средств состояний и переходов UML. Применяя их, вы можете специфицировать простой автомат, а это означает, что ваши поведенческие модели будут описаны лишь дугами (переходами) и вершинами (состояниями) – ничем более.

Автоматы UML включают множество средств, помогающих управлять сложными поведенческими моделями. Эти средства часто позволяют уменьшить количество необходимых состояний и переходов, а также смоделировать множество общих и иногда довольно сложных идиом, которые в противном случае пришлось бы отображать в виде простых автоматов. Некоторые из этих расширенных средств включают входные и выходные эффекты, внутренние переходы, деятельности и отложенные события; изображаются они в виде строк текста внутри пиктограммы состояния, как показано на рис. 22.4.

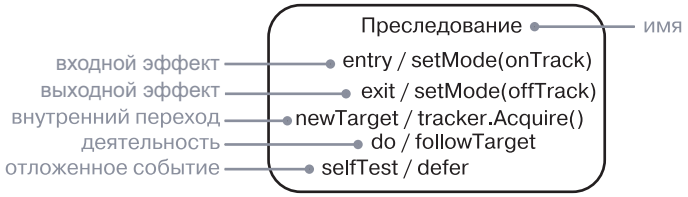


Рис. 22.4. Расширенные состояния и переходы

Входные и выходные эффекты

Иногда моделируемая ситуация требует выполнения некоторого установочного действия при входе в состояние – независимо от того, какой переход привел вас сюда. Аналогичным образом, когда объект покидает состояние, нужно выполнить некоторую очистку независимо от того, какой переход из него уводит. Например, в системе управления ракетой вы можете пожелать явно объявить эффект `onTrack` (ведетПреследование), когда система входит в состояние `Tracking` (Преследование), и `offTrack` (прекратилаПреследование) – когда покидает его. Используя простой автомат, можно достичь подобного эффекта, помещая эти действия соответственно на каждом входном и выходном переходе. Однако здесь велика вероятность ошибиться: необходимо постоянно помнить, что действия нужно создавать при добавлении каждого нового перехода. Более того, модификация действия означает, что будут затронуты все соседние переходы.

Как показано на рис. 22.4, UML предусматривает сокращение для этой идиомы. В пиктограмму состояния можно включить входной эффект (помеченный ключевым словом `entry`) и выходной эффект (помеченный ключевым словом `exit`), – каждый с указанием соответствующего действия. Всякий раз при входе в состояние вызывается входное действие, а при выходе из него – выходное.

Входной и выходной эффекты могут не иметь аргументов или защитных условий, но первый из них на верхнем уровне автомата класса может иметь параметры аргументов, которые автомат принимает при создании объекта.

Внутренние переходы

Внутри состояния часто возникает необходимость обрабатывать определенные события, не покидая его. В таком случае можно говорить о внутренних переходах, которые имеют тонкое отличие от переходов в себя. Когда дело касается перехода в себя (см. пример на рис. 22.3) событие инициирует переход, объект покидает состояние, выполняется некоторое действие (если оно есть) и затем объект вновь входит в то же состояние. Поскольку данный переход выходит и входит в состояние, он инициирует действие выхода, затем действие самого перехода и, наконец, действие входа.

Однако предположим, что вы хотите обработать событие, не выполняя при этом входные и выходные действия. UML предусматривает сокращение для этой идиомы в виде внутреннего перехода. *Внутренний переход* (internal transition) – это такой переход, который реагирует на событие соответствующим эффектом, не изменяя состояния. На рис. 22.4 событие `newTarget` (новаяЦель) помечает внутренний переход; если это событие происходит, когда объект находится в состоянии `Tracking` (Преследование), то при этом выполняется действие `tracker.Acquire()` (преследователь.Получить()), но состояние остается прежним, и никакого входного или выходного эффекта не наблюдается. Внутренний переход символизируется включением строки перехода (вместе с именем события, необязательным защитным условием и эффектом) в символ состояния вместо стрелки перехода. Отметим, что слова `entry` (вход), `exit` (выход) и `do` (действовать) являются ключевыми и не могут быть использованы в качестве имен событий. Всякий раз, когда объект находится в некотором состоянии и происходит событие, которым помечен внутренний переход, выполняется соответствующий эффект, но объект при этом не покидает состояния и не возвращается в него повторно. Таким образом, событие обрабатывается без вызова действий выхода и входа.

На заметку. Внутренние переходы могут включать события с параметрами и защитными условиями.

Деятельность внутри состояния

Когда объект находится в некоем состоянии, он обычно простаивает, ожидая наступления события. Однако иногда становится

необходимо смоделировать некую деятельность, выполняемую в этом состоянии (do-activity). Иными словами, пребывая в состоянии, объект может что-либо делать до тех пор, пока это не будет прервано событием. Например, пребывая в состоянии преследования – `Tracking`, объект может следовать за целью (`followTarget`). Как показано на рис. 22.4, в UML применяется специальный переход `do`, описывающий работу, которая должна выполняться внутри состояния после того, как будет выполнено входное действие. Можно также специфицировать поведение как последовательность действий – например, `do/op1(a);op2(b);op3(c)`. Если появление события вызовет переход, который приведет к выходу из состояния, любая выполняемая в состоянии деятельность немедленно прекратится.

На заметку. Деятельность внутри состояния – это эквивалент входного эффекта, который начинает деятельность при входе в состояние, и выходного эффекта, который останавливает ее при выходе из состояния.

Отложенные события

События обсуждаются в главе 21.

Рассмотрим такое состояние, как `Tracking` (Преследование). Как показано на рис. 22.3, предположим, что есть только один переход к этому состоянию, вызванный событием `contact` (контакт). Пока объект находится в состоянии `Tracking`, любые события, отличные от `contact`, а также от тех, которые обрабатываются подсостояниями, будут потеряны. Это значит, что событие может произойти, но будет проигнорировано и в результате не вызовет никаких действий.

В любой ситуации, связанной с моделированием, важно распознавать одни события и игнорировать другие. Первые вы включаете в модель как события, инициирующие переходы; вторые просто оставляете без внимания. Однако в определенных ситуациях необходимо принимать некоторые события и при этом откладывать реакцию на них на более позднее время. Например, пока объект находится в состоянии `Tracking`, придется откладывать реакцию на такие сигналы, как `selfTest` (самопроверка), посылаемый агентами поддержки работы системы.

В UML подобное поведение можно специфицировать, используя отложенные события. *Отложенное событие* (deferred event) – такое, обработка которого откладывается до тех пор, пока объект не перейдет в другое состояние. Если событие в этом новом состоянии уже не является отложенным, оно обрабатывается и может вызвать переход, как если бы оно произошло только что. Если автомат проходит через ряд состояний, в которых данное событие является отложенным, оно сохраняется до тех пор, пока не наступит состояние,

в котором оно перестанет считаться таковым. За этот период могут возникать другие события, не являющиеся отложенными. Как видно из рис. 22.4, отложенное событие можно отметить, добавив к его имени ключевое слово `defer`. В этом примере события `selfTest` могут иметь место в состоянии `Tracking`, но удерживаются до тех пор, пока объект не придет в состояние `Engaging` (Включение); тогда они обрабатываются так, будто произошли только что.

На заметку. Реализация отложенных событий требует наличия их внутренней очередности. Если происходит событие, которое помечено как отложенное, оно помещается в очередь. События извлекаются из нее, как только объект переходит в состояние, в котором они перестают считаться отложенными.

Вложенные автоматы

К автомату можно обратиться из другого автомата. Такие автоматы называются *вложенными* (submachines). Их удобно использовать, прорабатывая структуру крупных моделей состояний. Подробности приводятся в книге «UML» (выходные данные см. во введении, раздел «Цели»).

Подсостояния

Рассмотренные свойства состояний и переходов решают целый ряд типичных проблем моделирования автоматов. Однако у автоматов, рассматриваемых в UML, есть свойство, которое позволяет еще больше упростить моделирование сложного поведения, – *подсостояние* (substate), то есть такое состояние, которое входит в состав другого. Например, `Heater` (Обогреватель) может находиться в состоянии `Heating` (Обогрев) и в то же время – во вложенном состоянии `Activating` (Активация). В этом случае правильно будет сказать, что объект находится одновременно в состояниях `Heating` и `Activating`.

Составные состояния имеют вложенную структуру, подобную той, что встречается у композиций (см. главы 5 и 10).

Простое состояние – такое, которое не имеет внутренней структуры. Состояние, имеющее вложения, то есть подсостояния, называется *составным* (композитным). Подсостояния могут быть параллельными (ортогональными) или последовательными (неортогональными). В UML составное состояние изображается так же, как и простое, но с дополнительным разделом, в котором показан вложенный автомат. Глубина вложения состояний не ограничена.

Неортогональные (последовательные) подсостояния

Рассмотрим моделирование поведения банкомата. Система обслуживания может находиться в одном из трех основных состояний: `Idle` (Простой – ожидание взаимодействия с пользователем), `Active` (Активен – выполняет транзакцию, запрошенную пользователем) либо `Maintenance` (Обслуживание – возможно, пополняется запас наличности). В состоянии `Active` поведение банкомата описывается простой схемой: проверить счет пользователя, выбрать тип транзакции, выполнить транзакцию, напечатать чек. После этого банкомат снова возвращается в состояние `Idle`. Перечисленные этапы поведения можно представить как состояния `Validating` (Проверка), `Selecting` (Выбор), `Processing` (Обработка), `Printing` (Печать). Может быть, желательно настроить систему таким образом, чтобы она давала пользователю право выбрать и осуществить несколько транзакций после того, как выполнена проверка счета, но до печати общего чека.

Проблема в том, что на любом этапе у пользователя должна быть возможность отменить транзакцию и вернуть банкомат в состояние `Idle`. Такого эффекта можно достичь, используя простые автоматы, но это будет не очень красиво. Поскольку пользователь может отменить транзакцию в любой момент времени, пришлось бы включать соответствующий переход из любого состояния в последовательности `Active`. Это чревато ошибками, так как предусмотреть все необходимые переходы непросто, а наличие множества событий, прерывающих основной поток управления, приведет к появлению большого числа переходов из разных исходных состояний, которые будут оканчиваться в одном и том же целевом. При этом у каждого из них будут одни и те же инициирующее событие, защитное условие и действие.

При помощи вложенных подсостояний можно упростить моделирование этой задачи, как показано на рис. 22.5. Здесь у состояния `Active` есть внутренний автомат, в который входят подсостояния `Validating`, `Selecting`, `Processing`, `Printing`. Банкомат переходит из состояния `Idle` в состояние `Active`, когда пользователь вставляет в картоприемник кредитную карту. При входе в состояние `Active` выполняется действие `readCard` (читатьКарту). Начав с исходного состояния внутреннего автомата, управление переходит к состоянию `Validating`, затем к `Selecting` и, наконец, к `Processing`. После выхода из последнего управление может вернуться в состояние `Selecting` (если пользователь избрал другую транзакцию) или перейти в состояние `Printing`. Отсюда происходит безусловный завершающий переход в состояние `Idle`. Отметим, что у состояния `Active` есть действие выхода, которое обеспечивает автоматическое извлечение кредитной карты.

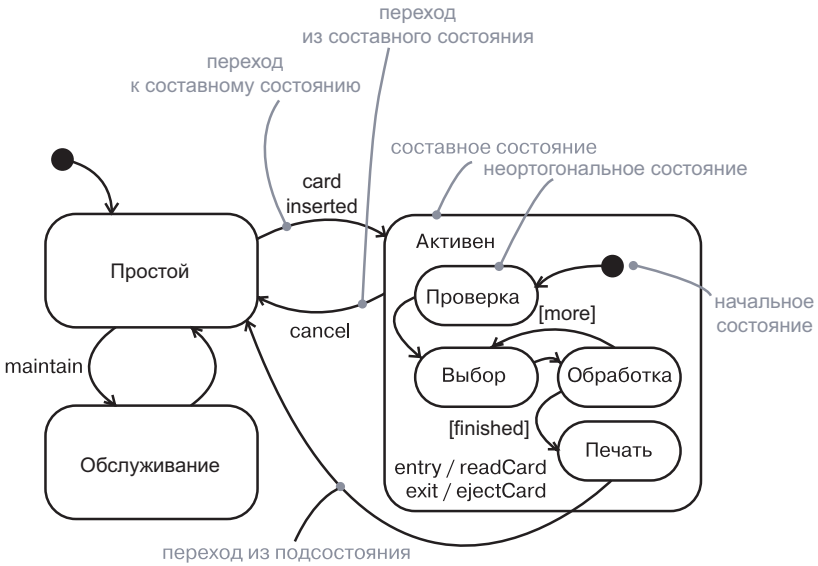


Рис. 22.5. Последовательные подсостояния

Также следует обратить внимание на переход из состояния `Active` в состояние `Idle`, инициируемый событием `cancel` (отмена). В любом подсостоянии состояния `Active` пользователь может отменить транзакцию и вернуть банкомат в состояние простоя (но только после извлечения кредитной карты из картоприемника – это действие, сопровождающее выход из состояния `Active`, происходит независимо от того, что послужило причиной выхода). Без подсостояний потребовался бы переход, инициируемый событием `cancel`, для каждого состояния подструктуры.

Такие подсостояния, как `Validating` и `Processing`, называются *неортогональными* (nonorthogonal), или *непересекающимися*. При наличии ряда неортогональных подсостояний в контексте включающего составного состояния объект может находиться одновременно в этом составном состоянии и только в одном из его подсостояний (или в конечном состоянии). Таким образом, неортогональные подсостояния разделяют пространство составного состояния на непересекающиеся состояния.

Переход от исходного состояния, внешнего по отношению к составному включающему состоянию, может вести либо к этому последнему, либо к одному из его подсостояний. Если целью перехода является составное состояние, то вложенный автомат должен иметь начальное состояние, которому передается управление после входа в составное состояние и после выполнения его входного действия, если таковое присутствует. Если цель перехода – вложенное подсостояние, то управление передается ему после выполнения входного

действия (если оно есть) составного состояния и (если оно есть) входного действия этого целевого подсостояния.

Переход, ведущий из составного состояния, может иметь в качестве исходного либо его само, либо одно из его подсостояний. В любом случае управление сначала покидает вложенное состояние (при этом выполняется его выходное действие, если таковое присутствует), а затем составное состояние (при этом выполняется его выходное действие, если оно есть). Переход, началом которого является составное состояние, по существу, прерывает деятельность вложенного автомата. Завершающий переход составного состояния выполняется, когда управление достигает его конечного подсостояния.

На заметку. Вложенные неортогональные автоматы могут иметь как максимум одно начальное подсостояние и одно конечное.

Исторические состояния

Автомат описывает динамическое поведение объекта, текущее состояние которого зависит от его прошлого. Автомат, по сути, специфицирует допустимый порядок состояний, которые может принимать объект за время своего жизненного цикла.

Если не указано иное, то когда переход приводит к составному состоянию, деятельность вложенного автомата начинается с его начального состояния (если только непосредственной целью перехода не является конкретное подсостояние). Однако иногда возникает необходимость моделировать объект, «помнящий» подсостояние, которое было активным перед тем, как он покинул составное состояние. Так, при моделировании поведения агента, выполняющего автоматическое резервное копирование на всех компьютерах сети, желательно помнить, на чем он остановился, когда, например, был прерван по запросу оператора.

Теоретически здесь допускается моделирование с помощью простого автомата, но это будет некрасиво. Нужно будет, чтобы каждое последовательное подсостояние посылало значение некоторой переменной, локальной по отношению к составному состоянию. Начальное состояние в этом составном должно будет иметь переход, снабженный защитным условием, проверяющим переменную, в каждое из своих подсостояний. Таким образом, при выходе из составного состояния можно будет запомнить последнее подсостояние, с тем чтобы при следующем входе в составное состояние сразу осуществить переход в него. Это неудобно, ведь вам придется помнить о том, что нужно «коснуться» всех подсостояний и установить

для каждого соответствующее выходное действие. В результате появится множество переходов из одного и того же начального состояния, ведущих к различным целевым подсостояниям с очень похожими (но все-таки разными) защитными условиями.

В UML предусмотрен более простой способ моделирования подобных идиом – с помощью исторических состояний. *Историческое состояние* (history state) позволяет составному состоянию, включающему неортогональные подсостояния, «помнить» подсостояние, которое было активным на момент последнего перехода из составного состояния вовне. Как показано на рис. 22.6, историческое состояние изображается в виде маленького кружочка с буквой «H» (History).

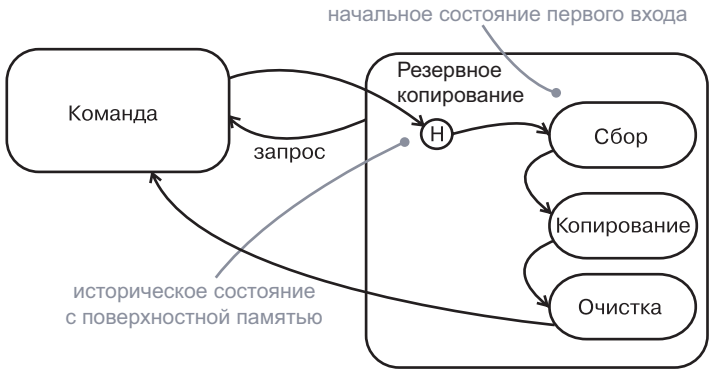


Рис. 22.6. Историческое состояние

Если нужно, чтобы переход активировал последнее подсостояние, то он показывается как ведущий из составного состояния вовне, непосредственно к историческому. Когда вы впервые входите в составное состояние, оно еще не имеет истории. Это означает, что в наличии имеется единственный переход от исторического состояния к последовательному подсостоянию, – в данном примере `Collecting` (Сбор).

Цель этого перехода специфицирует начальное состояние вложенного автомата при первом входе. Предположим, что во время состояния `BackingUp` (РезервноеКопирование) и `Copying` (Копирование) посылается событие `query` (запрос). Управление покидает `Copying` и `BackingUp` (вызывая при необходимости их выходные действия) и возвращается к состоянию `Command` (Команда). Когда завершится деятельность `Command`, завершающий переход вернется к историческому состоянию составного состояния `BackingUp`. На этот раз, поскольку существует история вложенного автомата, управление передается состоянию `Copying`, минуя состояние `Collecting`, так как именно `Copying` было последним активным подсостоянием перед последним выходом из `BackingUp`.

На заметку. Буквой «H» обозначается поверхностная память, которая хранит лишь последнее состояние ближайшего вложенного автомата. Можно специфицировать и глубинную память, изобразив ее в виде маленького кружочка с символами «H*» внутри. Такая история хранит состояния всех вложенных автоматов. Если имеется только один уровень вложенности, то поверхностная и глубинная память семантически эквивалентны. Если же уровней вложенности несколько, поверхностная память хранит состояние только ближайшего вложенного автомата, а глубинная – всех автоматов, вложенных друг в друга на любую глубину.

В любом случае, если вложенный автомат достиг завершающего состояния, то он теряет всю хранимую в нем историю и ведет себя так, как будто никакого входа в него еще не было.

Ортогональные подсостояния

Неортогональные подсостояния – это наиболее общий случай использования вложенных автоматов, с которым приходится сталкиваться на практике. Однако в некоторых ситуациях моделирования может понадобиться специфицирование ортогональных областей. Они позволяют описать несколько автоматов, работающих параллельно в контексте включающего объекта.

В примере на рис. 22.7 показано расширение состояния `Maintenance` (Обслуживание) из рис. 22.5. Оно декомпозируется на две ортогональные области – `Testing` (Тестирование) и `Commanding` (Обработка команд), показанные в виде вложений в `Maintenance`, но разделенные пунктирной линией. Каждая из этих ортогональных областей, в свою очередь, разбита на подсостояния. Когда управление передается из состояния `Idle` (Простой) в `Maintenance`, то общий поток разделяется на два параллельных – включающий объект будет одновременно пребывать в областях `Testing` и `Maintenance`. Более того, объект, находящийся в области `Commanding`, также пребывает и в состоянии `Waiting` (Ожидание) или `Command` (Команда).

На заметку. Неортогональные подсостояния отличаются от ортогональных по следующему признаку. Имея несколько неортогональных подсостояний на одном и том же уровне, объект может пребывать либо в одном из них, либо в другом. Имея же несколько ортогональных областей на одном и том же уровне, он будет одновременно пребывать в состоянии каждой из них.

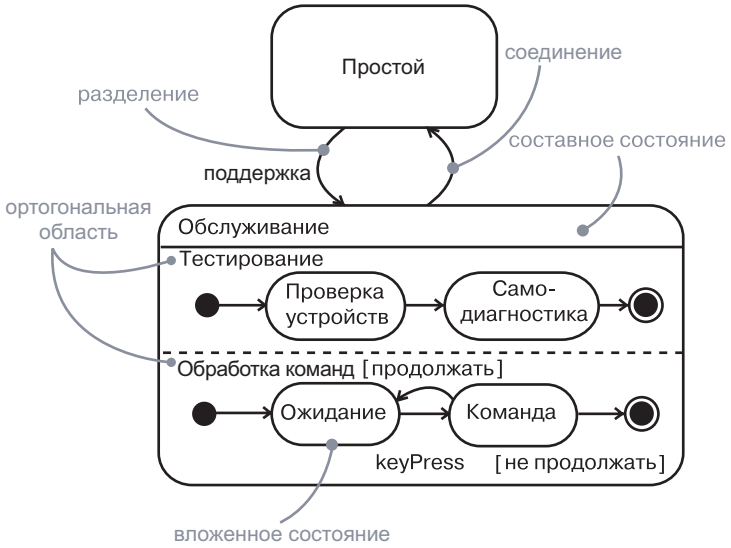


Рис. 22.7. Параллельные подсостояния

Прохождение по каждой из этих двух ортогональных областей осуществляется параллельно. В итоге каждый вложенный автомат достигает своего конечного состояния. Если одна ортогональная область переходит в конечное состояние раньше другой, управление в ней ожидает, пока другая область не достигнет своего конечного состояния. Когда же обе области оказываются в своих конечных состояниях, управление из них сливается обратно в общий поток.

Когда имеется переход в составное состояние, разбитое на ортогональные области, управление всегда разделяется на столько же параллельных потоков, сколько существует таких областей. Аналогично, при наличии перехода из составного состояния, разбитого на ортогональные области, параллельные потоки управления сливаются воедино. Это верно во всех случаях. Если все ортогональные области достигают своего конечного состояния или же присутствует явный переход из включающего составного состояния, управление сливается в один поток.

На заметку. Каждая ортогональная область может иметь начальное, конечное и историческое состояние.

Разделение и соединение

Обычно вход в составное состояние с ортогональными областями ведет к начальному состоянию каждой из этих областей. Но также возможно осуществить переход из внешнего состояния

непосредственно в одно или несколько ортогональных состояний. Такой процесс называется *разделением* (fork), поскольку управление передается от одного состояния нескольким ортогональным сразу. Графически это выражает толстая черная линия с одной входящей стрелкой и несколькими исходящими, каждая из которых указывает на ортогональное состояние. Если одна или несколько ортогональных областей не имеют целевых состояний, то неявно выбираются начальные состояния этих областей. Переход к единственному ортогональному состоянию внутри составного – также неявное разделение; начальные состояния всех других ортогональных областей являются неявными участниками разделения.

Аналогичным образом переход от любого состояния внутри составного с ортогональными областями инициирует выход из всех ортогональных областей. Такой переход часто представляет ошибочное условие, вызывающее прерывание всех параллельных вычислений.

Соединение (join) – это переход с несколькими входящими стрелками и одной исходящей. Каждая входящая стрелка должна исходить от состояния в разных ортогональных областях одного и того же составного состояния. Соединение может иметь инициирующее событие. Переход с соединением эффективен только тогда, когда активны все исходные состояния; статус других ортогональных областей составного состояния не важен. Если событие происходит, то управление покидает все ортогональные области составного состояния, а не только те, откуда ведут стрелки.

Рис. 22.8 – вариант предыдущего примера с явными переходами разделения и соединения. Переход `maintain` (поддержка) к составному

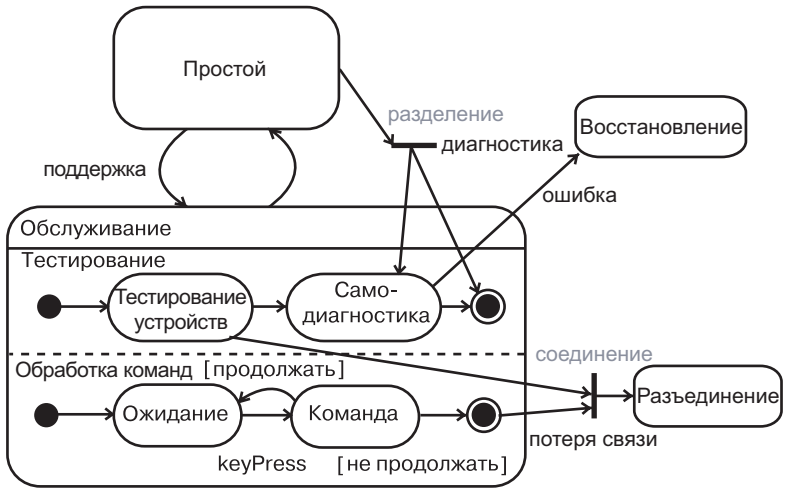


Рис. 22.8. Переходы с разделением и соединением

состоянию Maintenance (Обслуживание) – это по-прежнему неявное разделение, ведущее к начальным состояниям по умолчанию двух ортогональных областей. В этом примере, однако, присутствует переход с явным разделением от Idle (Простой) ко вложенным состояниям Self diagnose (Самодиагностика) и конечному состоянию области Commanding (Обработка команд). Конечное состояние – это явное состояние, которое может быть целью перехода. Если возникает событие ошибки во время Self diagnose, то происходит неявное слияние с переходом к Repair (Восстановление). При этом и Self diagnose, и любое активное состояние области Commanding завершается.

Есть также явный переход со слиянием к состоянию Offline (Разъединение). Он осуществляется только при возникновении события disconnect (потеря связи), когда активно и состояние Testing devices (Тестирование устройств), и конечное состояние области Commanding (Обработка команд). Если оба состояния неактивны, данное событие не имеет никакого эффекта.

Активные объекты

Активные объекты обсуждаются в главе 23.

Другой способ моделирования параллелизма – применение *активных объектов*. То есть вместо разбиения одного автомата объекта на несколько параллельных областей можно определить два активных объекта, каждый из которых реализует поведение одной из параллельных областей. Если на поведение одного из этих параллельных потоков влияет состояние другого, это можно смоделировать при помощи ортогональных областей. Если же поведение одного потока зависит от сообщений, посланных другим или другому, то имеет смысл воспользоваться активными объектами. Если между параллельными потоками происходит лишь минимальное общение либо вообще не наблюдается никакого, то в большинстве случаев применение активных объектов сделает ваш дизайн более наглядным.

Типичные приемы моделирования

Моделирование жизненного цикла объекта

Наиболее общее назначение автоматов – моделирование жизненного цикла объекта, в особенности если речь идет об экземплярах классов, вариантов использования и о системе в целом. В то время как взаимодействие моделирует поведение сообщества работающих вместе объектов, автомат моделирует поведение одного объекта в течение его жизненного цикла – например, так, как это

бывает с пользовательскими интерфейсами, контроллерами и различными техническими устройствами.

Когда моделируется жизненный цикл объекта, приходится явно специфицировать три момента: события, на которые объект может реагировать, непосредственную реакцию на эти события и влияние более раннего поведения на текущее. Кроме того, моделирование жизненного цикла объекта включает в себя принятие решений о порядке, в котором объект может осмысленно реагировать на события, начиная с момента его создания и заканчивая уничтожением.

Чтобы смоделировать жизненный цикл объекта, необходимо:

Объекты обсуждаются в главе 13, классы – в главах 4 и 9, варианты использования – в главе 17, системы – в главе 32, взаимодействия – в главе 16, кооперации – в главе 28, пред- и постусловия – в главе 10, интерфейсы – в главе 11.

- Определить контекст автомата – будет ли это класс или система в целом:
  – если речь идет о контексте класса или варианта использования, найти соседние классы, включая всех родителей данного класса и всех доступных ему по ассоциациям или зависимостям. Эти соседи – потенциальные цели действий, а также кандидаты на включение в защитные условия;
  – если же речь идет о контексте системы в целом, необходимо сосредоточить внимание на каком-то одном аспекте ее поведения. Теоретически каждый объект системы может участвовать в моделировании ее жизненного цикла, и за исключением работы с простейшими системами такие полные модели будут недоступны для понимания.
- Установить начальное и конечное состояния объекта. Чтобы управлять остальной моделью, возможно, понадобится установить пред- и постусловия начального и конечного состояний соответственно.
- Принять решения относительно событий, на которые должен реагировать объект. Их можно обнаружить в интерфейсах объекта (если они уже специфицированы); если же нет, следует рассмотреть, какие объекты могут взаимодействовать с данным в имеющемся контексте и какие события они могут передавать и принимать.
- От начального состояния до конечного выделить те состояния высшего уровня, в которых может находиться объект. Соединить их переходами, инициируемыми соответствующими событиями. Продолжать работу, добавляя действия к этим переходам.
- Идентифицировать входные и выходные действия (особенно если обнаружится, что соответствующая идиома используется в автомате).
- При необходимости расширить выделенные состояния подсостояниями.

- Проверить, все ли упомянутые события в автомате соответствуют событиям, ожидаемым в интерфейсе объекта. Аналогичным образом проверить, все ли события, ожидаемые интерфейсом объекта, обрабатываются в автомате. Наконец, рассмотреть места, где следует явно игнорировать события.
- Убедиться, что все действия, упомянутые в автомате, поддерживаются связями, методами и операциями включающего объекта.
- Провести трассировку автомата (либо вручную, либо с применением инструментальных средств), чтобы проверить его на соответствие ожидаемым последовательностям событий и реакций на них. Особенно старательно следует поискать недостижимые состояния и состояния, в которых автомат может зациклиться.
- После реорганизации автомата снова проверить его на предмет того, не изменилась ли семантика объекта.

В примере на рис. 22.9 представлен автомат контроллера домашней системы сигнализации, которая отвечает за отслеживание показаний разнообразных датчиков, расположенных по периметру дома.

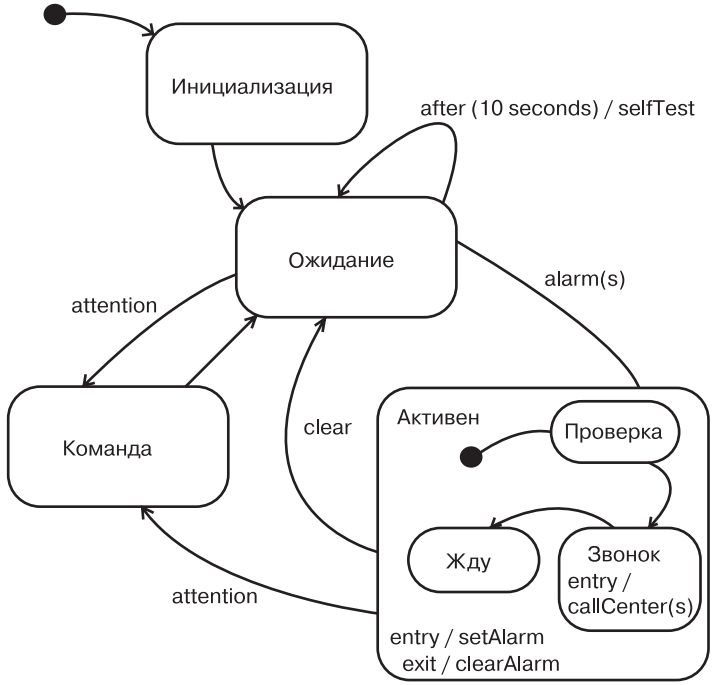


Рис. 22.9. Моделирование жизненного цикла объекта

В жизненном цикле этого контроллера имеются четыре основных состояния: `Initializing` (Инициализация – запуск работы), `Idle` (Простой – контроллер готов и ожидает сигналов тревоги или команд пользователя), `Command` (Команда – обработка команд пользователя) и `Active` (Активен – обработка сигнала тревоги). Впервые создаваемый объект контроллера сначала попадает в состояние `Initializing`, а затем безусловно переходит в состояние `Idle`. Подробности этих двух состояний не показаны, за исключением перехода в себя по истечении определенного периода времени (10 с) в состоянии `Idle`. Временные события подобного рода часто встречаются во встроенных системах, где присутствует таймер, вызывающий периодическую проверку работоспособности системы.

Управление передается от состояния `Idle` в состояние `Active` по получении события `alarm` (тревога). Последнее сопровождается параметром `s`, идентифицирующим датчик, который был задет. При входе в состояние `Active` в качестве входного выполняется действие `setAlarm` (поднятьТревогу) и управление передается сначала состоянию `Checking` (Проверка), затем состоянию `Calling` (Вызов), обеспечивающему вызов охранной компании для регистрации сигнала тревоги, и, наконец, состоянию `Waiting` (Ожидание). Состояния `Active` и `Waiting` завершаются только при «очистке» сигнала тревоги (`clearAlarm`) или же по инициативе пользователя, при посылке события `attention` (внимание), предположительно предшествующего команде.

Отметим, что конечного состояния здесь нет. И это опять же типично для встроенных систем, которые должны работать непрерывно.

Советы и подсказки

При моделировании автоматов на UML следует помнить, что каждый из них выражает динамические аспекты отдельного объекта, обычно представляющего экземпляр класса, варианта использования либо систему в целом. Хорошо структурированный автомат должен обладать следующими характеристиками:

- быть достаточно простым и не включать излишних состояний или переходов;
- благодаря ясному контексту иметь доступ ко всем объектам, видимым его включающему объекту (все эти соседи должны использоваться только при необходимости обеспечения поведения, специфицированного автоматом);
- быть эффективным, то есть реализовывать свое поведение с оптимальным балансом затрат времени и ресурсов.

Моделирование словаря системы обсуждается в главе 4.

- быть понятным, а потому именовать свои состояния и переходы в терминологии словаря системы;
- не допускать слишком глубоких уровней вложенности (вложенных подсостояний первого-второго уровней вполне хватит для описания самого сложного поведения);
- умеренно использовать ортогональные области, поскольку часто предпочтительно применение активных классов.

При изображении автомата в UML необходимо:

- избегать пересекающихся переходов;
- раскрывать составные состояния «по месту» только в том случае, если это сделает диаграмму более понятной.

Глава 23. Процессы и потоки

В этой главе:

Представления взаимодействия в контексте архитектуры программного обеспечения обсуждаются в главе 2.

Реальный мир суров и не прощает ошибок; вдобавок к этому в нем постоянно что-то случается. Различные события происходят одновременно. Поэтому при моделировании системы, предназначенной для работы в реальном мире, следует учитывать ее вид с точки зрения процессов, в котором основное внимание уделяется процессам и потокам, лежащим в основе механизмов параллелизма и синхронизации.

В UML каждый независимый поток управления моделируется как активный объект, описывающий процесс или поток и способный инициировать некоторое управляющее воздействие. Процесс – это ресурсоемкий поток управления, который выполняется параллельно с другими процессами; поток – это облегченный поток управления, выполняемый параллельно с другими потоками в рамках одного и того же процесса.

Построение абстракций таким образом, чтобы они могли безопасно функционировать при наличии нескольких потоков управления, – дело непростое. В частности, механизмы обмена информацией и синхронизации, которые при этом придется применять, существенно сложнее, чем для последовательных систем. Следует избегать как переусложнения модели (поскольку при наличии слишком большого числа параллельных потоков управления система начнет пробуксовывать), так и чрезмерного упрощения (потому, что недостаточная степень параллелизма не позволит оптимизировать пропускную способность системы).

Введение

Моделирование собачьей будки и небоскреба обсуждается в главе 1.

Для собаки, живущей в будке, распорядок дня прост и последователен. Едим. Спим. Гоняемся за кошкой. Снова едим. Мечтаем о том, как будем гоняться за кошкой. Забраться в будку, чтобы поспать или укрыться от дождя, не представляет проблемы, поскольку кроме собаки ни у кого не возникнет потребность воспользоваться входом. Никакой конкуренции за ресурсы.

Семейные заботы не столь просты. Не впадая в философствование, отметим, что каждый член семьи живет своей собственной жизнью, но в то же время взаимодействует с другими домочадцами (вместе обедают, смотрят телевизор, играют, проводят уборку). Члены семьи совместно пользуются некоторыми ресурсами. Порой у детей бывает общая спальня, на всю семью может быть только один телефон или компьютер. Родственники распределяют между собой обязанности: отец ходит в прачечную и в бакалейную лавку, мать оплачивает счета и работает в саду, дети помогают убираться в доме и готовить. Борьба за использование общих ресурсов и координация домашних обязанностей становятся предметом споров. Наличие одной ванной комнаты в ситуации, когда все одновременно собираются в школу, – это проблема, а обед не будет готов, если папа не сходил в магазин.

Но по-настоящему сложна жизнь небоскреба и его арендаторов. Сотни, а то и тысячи людей приходят в одно и то же здание, и у каждого свои планы. Все должны пройти через определенное количество подъездов. Все пользуются одними и теми же лифтами, а их число тоже не бесконечно. Посетителей обслуживают одни и те же системы отопления, кондиционирования, водоснабжения, канализации и подачи электроэнергии. На всех одна парковка. Если люди хотят работать слаженно, то должны общаться и синхронизировать свои действия.

Объекты обсуждаются в главе 13.

В UML каждый независимый поток управления моделируется в виде активного объекта. *Активный объект* – это процесс или поток, способные инициировать управляющее воздействие. Как и любой другой объект, активный объект может быть экземпляром класса. В таком случае говорят, что он является экземпляром *активного класса*. Подобно прочим объектам, активные объекты могут общаться друг с другом, посылая сообщения, хотя в данном случае передача сообщений должна быть дополнена семантикой параллелизма, чтобы облегчить взаимодействие независимых потоков управления.

Многие языки программирования непосредственно поддерживают концепцию активного объекта. В языки Java, Smalltalk и Ada параллелизм встроен. C++ поддерживает параллелизм за счет различных библиотек, в основе которых лежат механизмы параллелизма, обеспечиваемые операционной системой. Применение UML для визуализации, специфицирования, конструирования и документирования этих абстракций необходимо потому, что без него почти невозможно рассуждать о параллелизме, обмене информацией и синхронизации.

Классы обсуждаются в главе 4, сигналы – в главе 21.

Графическое представление активного класса, принятое в UML, вы видите на рис. 23.1. Активные классы – это разновидность классов, поэтому их нотация включает все соответствующие разделы – для имени класса, атрибутов и операций. Активные классы часто получают сигналы, которые обычно перечисляются в дополнительном разделе.

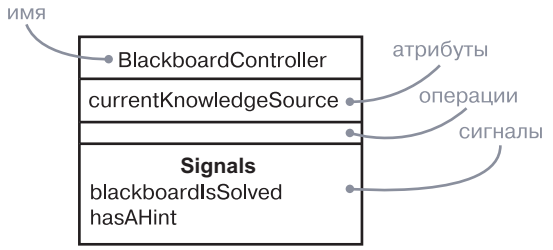


Рис. 23.1. Активный класс

Базовые понятия

Диаграммы взаимодействия обсуждаются в главе 19.

Активный объект (active object) – это объект, который владеет процессом или потоком и может инициировать управляющее воздействие.

Активный класс (active class) – это класс, экземплярами которого являются активные объекты. Изображается в виде прямоугольника с утолщенными боковыми границами (см. рис. 23.1).

Процесс (process) – это ресурсоемкий поток управления, который может выполняться параллельно с другими процессами.

Поток (thread) – это облегченный поток управления, который может выполняться параллельно с другими потоками в рамках одного процесса. Процессы и потоки изображаются в виде активных классов со стереотипами, а на диаграммах взаимодействия часто выступают в роли последовательностей.

Поток управления

В строго последовательной системе имеется только один поток управления. Это означает, что в каждый момент времени выполняется одно и только одно действие. При запуске последовательной программы

Действующие лица обсуждаются в главе 17.

управление переходит к точке входа в программу, после чего выполняется одна операция за другой. Даже внешние по отношению к системе действующие лица функционируют параллельно, последовательная программа будет обрабатывать по одному событию за раз, отбрасывая тем самым одновременные с ним события.

Действия обсуждаются в главе 16.

Поэтому такая последовательность и называется *потоком управления*. Если трассировать выполнение последовательной программы, то точка выполнения будет перемещаться из одного предложения к другому – последовательно. Встречаются, конечно, ветвления, циклы, переходы, а при наличии рекурсии или итерации – движения потока вокруг одной точки. Но, несмотря на все это, в последовательной системе есть только один поток выполнения.

В параллельной же системе потоков управления несколько, а значит, в один и тот же момент времени имеет место различная деятельность. Каждый из нескольких одновременно выполняемых потоков управления начинается с точки входа в некоторый процесс или поток. Если сделать моментальный снимок параллельной системы во время ее работы, то мы увидим несколько точек управления (по крайней мере, логических).

Узлы обсуждаются в главе 27.

Активный класс в UML используется для представления процесса или потока, в контексте которого выполняется независимый поток управления, работающий параллельно с другими, пользующимися равными с ним правами.

На заметку. Истинного параллелизма можно достичь тремя способами: во-первых, распределяя активные объекты между несколькими узлами, во-вторых, помещая активные объекты на узлы с несколькими процессорами, и, в-третьих, комбинируя оба метода.

Классы и события

Классы обсуждаются в главе 4 и 9.

Активные классы – это именно классы, хотя и обладающие весьма специфическим свойством. Активный класс представляет независимый поток управления, тогда как обычный класс не связан с таковым. В отличие от активных, обычные классы неформально называют пассивными, так как они неспособны инициировать независимое управляющее воздействие.

Объекты обсуждаются в главе 13, атрибуты и операции – в главе 4, связи – в главах 4 и 10, механизмы расширения – в главе 6, интерфейсы – в главе 11.

Активные классы применяются для моделирования семейств процессов или потоков. На практике это означает, что активный объект – экземпляр активного классификатора – материализует процесс или поток. При моделировании параллельных систем с помощью активных объектов вы присваиваете имя каждому

независимому потоку управления. В результате создания активного объекта запускается ассоциированный с ним поток управления; после уничтожения объекта этот поток завершается.

Активные классы обладают теми же свойствами, что и любые другие классы. Они могут иметь экземпляры, атрибуты и операции, а также принимать участие в связях зависимости, обобщения и ассоциации (включая агрегации). Активные классы вправе пользоваться предоставляемыми UML механизмами расширения, в том числе стереотипами, помеченными значениями и ограничениями. Встречаются активные классы – реализации интерфейсов. Активные классы могут реализовываться посредством кооперации, а их поведение может специфицироваться с помощью автоматов. Допускается участие активных классов в кооперациях.

Автоматы обсуждаются в главе 22, события – в главе 21.

На диаграммах активные объекты встречаются там же, где и пассивные. Можно моделировать кооперации активных и пассивных объектов с помощью диаграмм взаимодействия (включая диаграммы последовательности и коммуникации). Активный объект может выступать в роли целевого объекта события в автомате.

Если же говорить об автоматах, то как активные, так и пассивные объекты могут посылать и получать события сигналов и вызовов.

На заметку. По нашему мнению, использование активных классов необязательно – они не привносят ничего существенного в семантику.

Коммуникация

Взаимодействия обсуждаются в главе 16.

Кооперирующиеся между собой объекты взаимодействуют путем обмена сообщениями. В системе, где есть одновременно активные и пассивные объекты, следует рассматривать четыре возможных комбинации.

Во-первых, сообщение может быть передано одним пассивным объектом другому такому же. Если предположить, что в любой момент времени существует лишь один поток управления, проходящий через оба объекта, такое взаимодействие – не что иное, как простой вызов операции.

Сигналы событий и события вызова обсуждаются в главе 21.

Во-вторых, сообщение может быть передано от одного активного объекта другому активному. Здесь можно говорить о *межпроцессной коммуникации*, которая может осуществляться двумя способами. В первом варианте некоторый активный объект может синхронно вызывать операцию другого. Такой способ имеет семантику рандеву: вызывающий объект затребует выполнение операции и ждет, пока принимающая сторона получит вызов, выполнит некоторую операцию

и вернет некоторый объект (если есть что возвращать); затем оба объекта продолжают работать независимо друг от друга. В течение всего времени выполнения вызова оба потока управления будут блокированы. Во втором варианте один активный объект может асинхронно послать сигнал другому или вызвать его операцию. Семантика такого способа напоминает почтовый ящик (mailbox): вызывающая сторона посылает сигнал или вызывает операцию, после чего продолжает работу. Тем временем получающая сторона принимает сигнал или вызов, как только будет к этому готова. Пока она обрабатывает запрос, все вновь поступающие события или вызовы ставятся в очередь. Отреагировав на запрос, принимающий объект продолжает свою работу. Семантика почтового ящика проявляется в том, что оба объекта не синхронизированы – просто один оставляет сообщение для другого.

В UML синхронное сообщение изображается закрашенной стрелкой, а асинхронное – обычной (см. рис. 23.2).

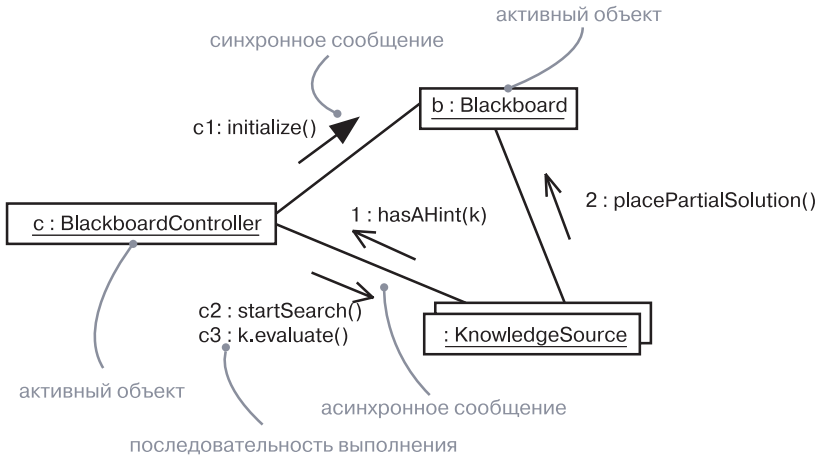


Рис. 23.2. Коммуникация

В-третьих, сообщение может быть передано от активного объекта пассивному. Трудности возникают в случае, когда сразу несколько активных объектов передают свой поток управления одному и тому же пассивному. В такой ситуации следует очень аккуратно моделировать синхронизацию потоков, о чем мы поговорим в следующем разделе.

Ограничения обсуждаются в главе 6.

В-четвертых, пассивный объект может передавать сообщения активному. На первый взгляд это может показаться некорректным, но если вспомнить, что каждый поток управления принадлежит некоторому активному объекту, то становится ясно, что передача пассивным объектом сообщения активному имеет

ту же семантику, что и обмен сообщениями между двумя активными объектами.

На заметку. С помощью ограничений можно моделировать различные вариации посылки синхронных и асинхронных сообщений. Например, для моделирования отложенного рандеву, имеющегося в языке Ada, можно воспользоваться синхронным сообщением с ограничением в формате `{wait=0}`, которое говорит о том, что вызывающий объект не будет дожидаться получателя. Можно смоделировать тайм-аут с помощью ограничения в формате `{wait=1ms}`, которое говорит, что вызывающий объект будет ждать приема сообщения получателем не более одной миллисекунды.

Синхронизация

Попробуйте на секунду представить себе многочисленные потоки управления в параллельной системе. Когда поток проходит через некоторую операцию, мы говорим, что эта операция является точкой выполнения. Если операция определена в некотором классе, то можно сказать, что точкой выполнения является конкретный экземпляр этого класса. В одной операции (и, стало быть, в одном объекте) могут одновременно находиться несколько потоков управления, а бывает и так, что разные потоки находятся в разных операциях, но все же в одном объекте.

Проблема возникает тогда, когда в одном объекте находятся сразу несколько потоков управления. Если не проявить осторожность, то более чем один поток может модифицировать один и тот же атрибут, что приведет к некорректному изменению состояния объекта или потере информации. Это классическая проблема взаимного исключения. Ошибки при обработке такой ситуации могут стать причиной различных видов конкуренции между потоками и их взаимной интерференции, что проявляется в таинственных и не подающихся воспроизведению сбоях параллельной системы.

Ключ к решению проблемы – сериализация доступа к критическому объекту. У данного подхода есть три разновидности, суть каждой из которых заключается в присоединении к операциям, определенным в классе, некоторых синхронизирующих свойств. UML позволяет моделировать все три возможности:

1. `Sequential` (последовательная) – вызывающие стороны должны координировать свои действия еще до входа в вызываемый объект, так что в любой момент времени внутри объекта находится ровно один поток управления. При наличии

нескольких потоков управления не могут гарантироваться семантика и целостность объекта;

2. `Guarded` (защищенная) – семантика и целостность объекта гарантируются при наличии нескольких потоков управления путем упорядочения вызовов всех защищенных операций объекта. По существу, в каждый момент времени может выполняться одна операция над объектом, что сводит такой подход к последовательному. При этом существует опасность взаимной блокировки;
3. `Concurrent` (параллельная) – семантика и целостность объекта при наличии нескольких потоков управления гарантируется за счет отделения операций изменения данных от операций чтения. Это достигается благодаря тщательному соблюдению правил проектирования.

Некоторые языки программирования поддерживают перечисленные конструкции непосредственно. Так в языке Java есть свойство `synchronized`, эквивалентное свойству `concurrent` в UML. В любом языке, поддерживающем параллельность, все три подхода можно реализовать с помощью *семафоров* (semaphores).

Ограничения обсуждаются в главе 6.

На рис. 23.3 показано, как эти свойства присоединяются к операции, – путем применения нотации, принятой в UML для ограничений. Обратите внимание, что одновременность должна быть объявлена отдельно как для каждой операции, так и для целого объекта. Объявление одновременности для операции означает беспроблемное единовременное выполнение ее многочисленных вызовов. Объявление одновременности для объекта позволяет вызовам разных операций выполняться одновременно и без ошибок. Это более строгое условие.

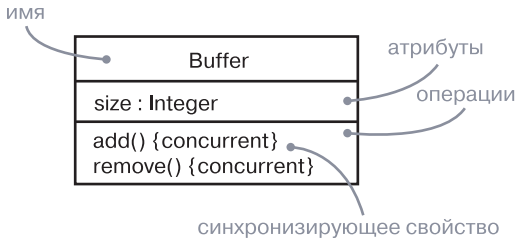


Рис. 23.3. Синхронизация

На заметку. С помощью ограничений можно моделировать различные вариации примитивов синхронизации. Например, можно модифицировать свойство `concurrent`, разрешив наличие нескольких читателей, но только одного писателя.

Типичные приемы моделирования

Моделирование множества потоков управления

Механизмы обсуждаются в главе 29, диаграммы классов – в главе 8, диаграммы взаимодействия – в главе 19.

Представления процессов обсуждаются в главе 19, классы – в главах 4 и 9, связи – в главах 5 и 10.

Построение системы с несколькими потоками управления – непростая задача. Надо не только решить, как распределить работу между параллельными активными объектами, но и продумать механизмы коммуникации и синхронизации между активными и пассивными объектами в системе, гарантирующие правильность их поведения в присутствии нескольких потоков управления. Поэтому полезно визуализировать способы взаимодействия этих потоков. В UML это можно сделать с помощью диаграмм взаимодействия (для описания динамической семантики), в которых участвуют активные классы и объекты.

Для моделирования нескольких потоков управления потребуется:

- Идентифицировать возможности распараллеливания действий и материализовать каждый поток управления в виде активного класса. Сгруппировать общие множества активных объектов в активный класс. Важно избегать усложнения системы вследствие слишком большого количества параллельных потоков управления.
- Рассмотреть баланс распределения обязанностей между этими активными классами, а затем исследовать, с какими другими активными и пассивными классами статически кооперируется каждый из них. Убедиться, что каждый активный класс имеет внутреннюю структуру с высокой степенью сцепления и слабо связан с соседними классами, и что для каждого класса правильно выбран набор атрибутов, операций и сигналов.
- Отобразить статические решения в виде диаграмм классов, явно выделив каждый активный класс.
- Рассмотреть, как каждая группа классов динамически кооперируется с прочими. Отобразить свои решения на диаграммах взаимодействия. Явно показать активные объекты как начальные точки соответствующих потоков управления. Идентифицировать каждую связанную последовательность, присваивая ей имя активного объекта.
- Обратить особое внимание на коммуникации между активными объектами, по мере необходимости пользуясь как синхронными, так и асинхронными сообщениями.

- Обеспечить синхронизацию активных объектов и тех пассивных объектов, с которыми они кооперируются. Применяйте наиболее подходящую семантику – последовательную, защищенную или параллельную.

На рис. 23.4 показана часть представления трейдерской системы с точки зрения процессов. Вы видите три объекта, которые параллельно питают систему информацией: StockTicker (БиржевойТикер), IndexWatcher (НаблюдательИндекса) и CNNNewsFeed (НовостнаяЛентаCNN), названные соответственно s, i и c. Два из них, s и I, обмениваются каждый со своим экземпляром класса Analyst (Аналитик) – a1 и a2. В рамках этого небольшого фрагмента модели класс Analyst может быть спроектирован в упрощенном виде – из расчета на то, что в каждый момент времени в любом из его экземпляров может быть активен только один поток управления. Однако оба экземпляра класса Analyst одновременно общаются с объектом AlertManager (ДиспетчерОповещений), которому мы дали имя m. Следовательно, m необходимо спроектировать так, чтобы он сохранял свою семантику в присутствии нескольких потоков управления. Объекты m и c одновременно общаются с t – объектом класса TradingManager (МенеджерПоПродажам). Каждому соединению присвоен порядковый номер, определяемый тем, какой поток управления им владеет.

На заметку. Диаграммы взаимодействия, подобные представленной выше, полезны для визуализации тех мест, где два потока управления могут пересекаться и где, следовательно, необходимо обратить особое внимание на проблемы коммуникации и синхронизации. С помощью инструментальных средств допускается реализация дополнительных визуальных меток – например, раскрашивание потоков в разные цвета.

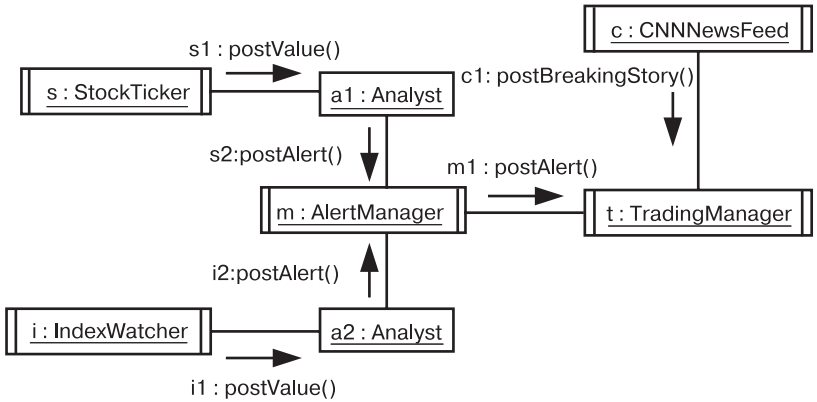


Рис. 23.4. Моделирование потоков управления

Автоматы обсуждаются в главе 22

К подобным диаграммам часто присоединяют автоматы с ортогональными состояниями, чтобы в деталях передать поведение активных объектов.

Моделирование межпроцессных коммуникаций

Сигналы и события вызова обсуждаются в главе 21.

При включении в систему нескольких потоков управления необходимо также рассмотреть механизмы, с помощью которых объекты из разных потоков управления взаимодействуют друг с другом. Объекты, находящиеся в разных потоках (существующих в рамках одного и того же процесса), могут общаться с помощью событий, сигналов или вызовов, причем последние могут иметь синхронную и асинхронную семантику. Для обмена информацией через границы процессов, у каждого из которых свое собственное адресное пространство, обычно применяются другие механизмы.

Моделирование месторасположения обсуждается в главе 24.

Сложность проблемы межпроцессной коммуникации усугубляется еще и тем, что в распределенных системах процессы могут выполняться на различных узлах. Существует два классических подхода к межпроцессной коммуникации: передача сообщений и вызовы удаленных процедур. В UML эти механизмы моделируются как асинхронные и синхронные события соответственно. Но поскольку это уже не простые вызовы внутри одного процесса, то проект необходимо обогатить дополнительной информацией.

Стереотипы и примечания обсуждаются в главе 6, кооперации – в главе 28.

Для моделирования межпроцессной коммуникации необходимо:

- Смоделировать несколько потоков управления.
- Смоделировать обмен сообщениями с помощью асинхронных коммуникаций, а вызовы удаленных процедур – с помощью синхронных.
- Неформально описать используемый механизм с помощью примечаний или, более строго, с помощью кооперации.

На рис. 23.5 показана распределенная система бронирования билетов, в которой процессы выполняются в четырех узлах. Каждый объект маркирован стереотипом process, а также помеченным значением location, которое определяет его физическое положение. Коммуникации между объектами ReservationAgent (АгентБронирования), TicketingManager (ДиспетчерВыдачиБилетов) и HotelAgent (ГостиничныйАгент) асинхронны. В примечании написано, что коммуникации построены на основе службы сообщений, реализованной на JavaBeans. Коммуникация между объектами TripPlanner (ПланировщикМаршрута) и ReservationSystem (СистемаРезервирования) синхронная. Семантика их взаимодействия показана в кооперации,

названной CORBA ORB. TripPlanner выступает в роли клиента, а ReservationAgent – сервера. Раскрыв эту кооперацию, вы увидите детали совместной работы сервера и клиента.

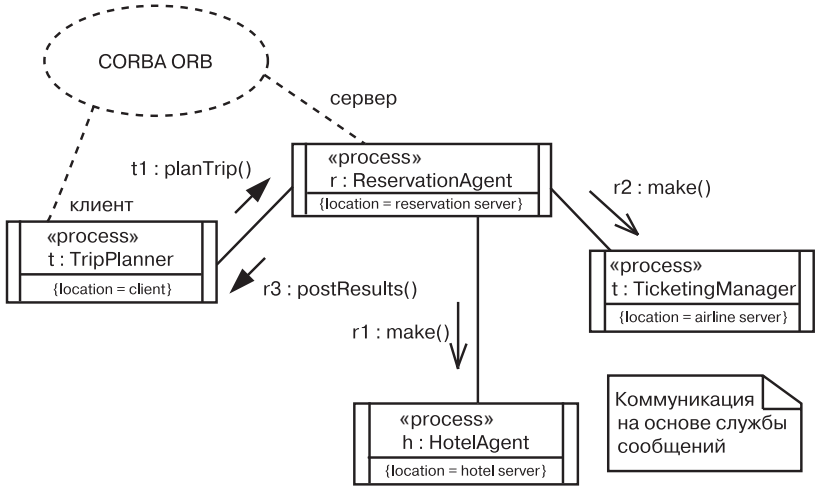


Рис. 23.5. Моделирование межпроцессной коммуникации

Советы и подсказки

Хорошо структурированные активный класс и активный объект обладают следующими свойствами:

- представляют независимый поток управления, который максимально использует возможности истинного параллелизма в системе;
- не настолько мелки, чтобы требовать наличия других активных элементов, избыток которых может создать переусложненную и нестабильную архитектуру процессов;
- аккуратно управляют коммуникацией между активными элементами, выбирая наиболее подходящий случаю механизм – синхронный или асинхронный;
- исходят из трактовки каждого объекта как критической области, используя подходящие синхронизирующие свойства для сохранения его семантики в присутствии нескольких потоков управления.

Рисуя в UML активный класс или активный объект, руководствуйтесь следующими принципами:

- показывайте только те атрибуты, операции и сигналы, которые важны для понимания абстракции в соответствующем контексте;
- явно показывайте все синхронизирующие свойства операции.

Глава 24. Время и пространство

В этой главе

- Время, длительность и местоположение
- Моделирование временных ограничений
- Моделирование распределения объектов
- Моделирование мигрирующих объектов
- Системы реального времени и распределенные системы

Как уже говорилось, реальный мир суров и не прощает ошибок. События в нем происходят в непредсказуемые моменты времени и, тем не менее, требуют адекватной и своевременной реакции. Системные ресурсы могут быть распределены по всему миру, а некоторые способны перемещаться в пространстве, – в связи с этим возникают проблемы задержек, синхронизации, безопасности и качества услуг.

Моделирование времени и пространства – важный элемент любой распределенной системы или системы реального времени. Для визуализации, специфицирования, конструирования и документирования таких систем UML предлагает различные средства, включая отметки времени, временные выражения, ограничения и помеченные значения.

Проектирование систем реального времени и распределенных систем – трудная задача. Хорошая модель должна высветить все необходимые и достаточные пространственно-временные свойства системы.

Введение

Приступая к моделированию большинства программных систем, обычно делают установку на идеальность среды: предполагают, что доставка сообщения осуществляется мгновенно, сетевые сбои исключены, рабочие станции никогда не выходят из строя, а загрузка сети всегда равномерна. К сожалению, реальность отнюдь не укладывается в эту схему: сообщения доставляются с некоторой задержкой (а иногда и вовсе не доставляются), сеть «падает»,

рабочие станции «зависают» и загрузка сети далека от сбалансированной. Поэтому, рассматривая моделируемую систему, необходимо принимать во внимание как пространственные, так и временные характеристики.

Система реального времени (real time system) называется так потому, что она должна выполнять свои функции в строго определенный абсолютный или относительный момент времени и затрачивать на это предсказуемое и зачастую ограниченное время. Среди подобных систем бывают такие, для которых требуемое время реакции исчисляется нано- или миллисекундами. Но встречаются системы «почти реального времени», для которых допустимое время реакции – порядка секунды и даже больше.

Компоненты обсуждаются в главе 26, узлы – в главе 27.

Под *распределенной системой* (distributed system) понимается такая система, компоненты которой могут быть физически размещены на различных узлах. Узлы могут представлять собой разные процессоры, смонтированные в одном и том же корпусе, или разные компьютеры, находящиеся в противоположных точках земного шара.

Чтобы удовлетворить потребности моделирования систем реального времени и распределенных систем, UML включает графическое представление для отметок времени, временных выражений, временных ограничений и местоположения, как показано на рис. 24.1.

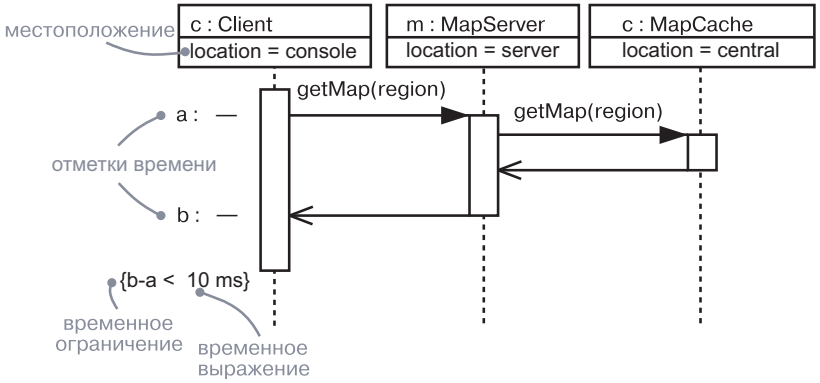


Рис. 24.1. Временные ограничения и местоположение

Базовые понятия

Отметка времени (timing mark) служит для обозначения момента времени, в который произошло событие. Она изображается в виде засечки (короткой горизонтальной линии) на границе диаграммы последовательности.

Временное выражение (time expression) – это выражение, значением которого является абсолютное или относительное время. Временное выражение может быть также сформировано с использованием имени сообщения и указания стадии его обработки, например `request.sendTime` или `request.receiveTime`.

Временное ограничение (timing constraint) – это семантическое утверждение об относительном или абсолютном времени. Графически временное ограничение изображается как любое другое – строкой, заключенной в скобки, и обычно связано с некоторым элементом зависимостью.

Местоположение (location) – это размещение компонента в узле. Является атрибутом объекта.

Время

События, включая события времени, обсуждаются в главе 21, сообщения и взаимодействия – в главе 16, временные ограничения – в главе 6.

Для систем реального времени, как следует из самого названия, важна своевременность реакции. События в них могут происходить регулярно или спонтанно, но в любом случае время реакции на событие должно быть предсказуемо – либо в абсолютном выражении, либо относительно момента возникновения события.

Передача сообщений – это один из динамических аспектов любой системы, поэтому при моделировании ее временных особенностей с помощью UML можно каждому сообщению, принимающему участие во взаимодействии, дать имя, которое будет использоваться как отметка времени. Обычно сообщениям, принимающим участие во взаимодействии, имена не присваиваются. Как правило, при их изображении используется имя события, например сигнала или вызова. При этом нельзя вводить имя события в запись выражения, поскольку одно и то же событие может повлечь за собой различные сообщения. Если имеет место такого рода неоднозначность, следует присвоить явное имя сообщению в отметке времени, чтобы в дальнейшем его можно было использовать во временном выражении. Если задано имя сообщения, допускается применение любой из трех его функций – `sendTime`, `receiveTime`, `transmissionTime`. (Эти функции – наше собственное изобретение, они не описаны в UML. Системы реального времени могут включать и другие функции.) Для синхронных вызовов можно также указать ссылку на время *кругового* (round-trip) сообщения – `executionTime` (еще одно наше нововведение). Эти функции удобны для построения сложных временных выражений (в том числе включающих веса и смещение), которые представляют собой константы или переменные (если эти переменные можно вычислить в момент выполнения). Наконец, как показано на рис. 24.2, временное выражение можно поместить внутрь временного ограничения, чтобы описать поведение системы

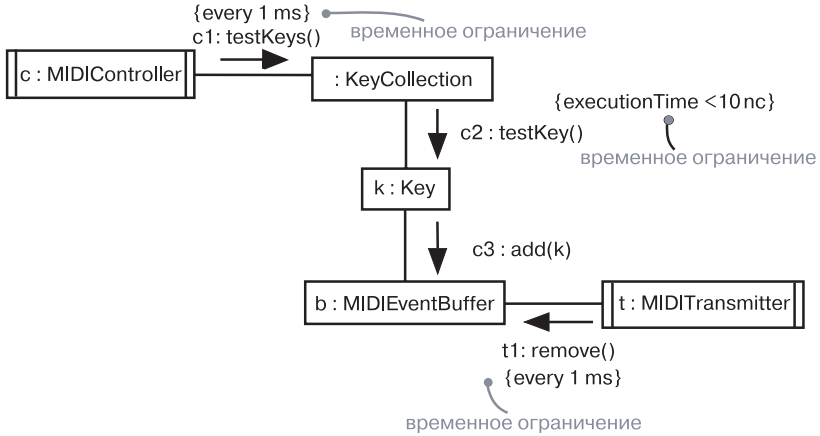


Рис. 24.2. Время

во времени. Как и любые другие ограничения, они изображаются рядом с соответствующим сообщением или явно присоединяются с помощью связей зависимости.

На заметку. Вместо явного упоминания времени в выражениях стоит употреблять именованные константы, в особенности при моделировании сложных систем. Константы можно определить в одной части модели, а затем использовать в других местах. Тогда будет проще модифицировать модель, если временные требования к системе изменятся.

Местоположение

Компоненты обсуждаются в главе 15, узлы – в главе 27, диаграммы размещения – в главе 31, дихотомия классов и объектов рассматривается в главах 2 и 13, классы – в главах 4 и 9.

Распределенные системы по своей природе состоят из компонентов, физически рассредоточенных по разным узлам. Очень часто *местоположение* (location) компонентов фиксируется в момент установки системы. Но встречаются и такие системы, в которых компоненты мигрируют с одного узла на другой.

В UML вид системы с точки зрения размещения моделируется с помощью диаграмм размещения, описывающих топологию процессоров и устройств, на которых функционирует система. Артефакты, такие как исполняемые модули, библиотеки и таблицы, размещаются в этих узлах. Каждый экземпляр узла владеет собственными экземплярами тех или иных артефактов, а каждый экземпляр артефакта принадлежит ровно одному экземпляру узла (хотя различные экземпляры артефакта одного вида могут находиться на разных узлах).

Компоненты и классы могут быть материализованы в виде артефактов. Так, на рис. 24.3 класс `LoadAgent` (АгентЗагрузки) материализован

в виде артефакта `initializer.exe`, который размещается на узле типа `Router` (Маршрутизатор).

Как видно из рисунка, моделировать положение артефакта в UML можно двумя способами. Во-первых, как в случае с `Router`, можно физически поместить элемент (его текстовое или графическое описание) в дополнительный раздел включающего узла. Во-вторых, вы можете использовать зависимость с ключевым словом `«deploy»` от артефакта к узлу, который его содержит.

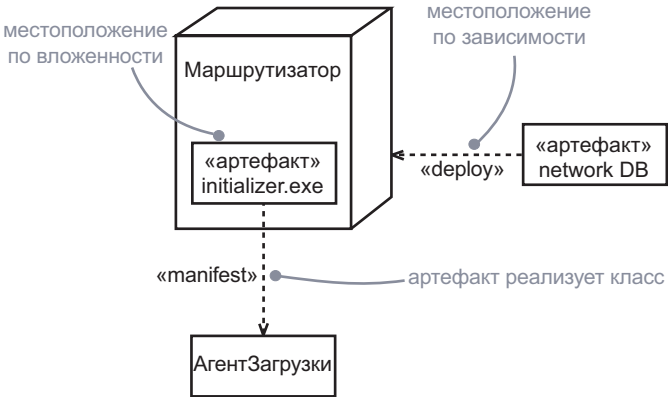


Рис. 24.3. Местоположение

Типичные приемы моделирования

Моделирование временных ограничений

Ограничения – это один из механизмов расширения UML, рассматриваемый в главе 6.

Абсолютное время события и относительное время между событиями – вот основные временные аспекты систем реального времени, при моделировании которых находят применение временные ограничения.

Для моделирования временных ограничений понадобится:

- Для каждого события во взаимодействии рассмотреть, должно ли оно начинаться в определенный абсолютный момент времени. Промоделировать это свойство с помощью временного ограничения на сообщение.
- Для каждой представляющей интерес последовательности сообщений во взаимодействии рассмотреть, ограничено ли время ее выполнения. Промоделировать это свойство с помощью временного ограничения на последовательность.

Например, левое ограничение на рис. 24.4 специфицирует начальное время повторяющегося события вызова `refresh`. Временное

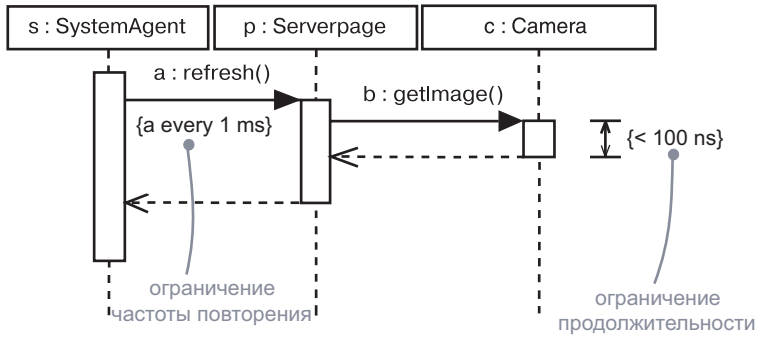


Рис. 24.4. Моделирование временных ограничений

ограничение, находящееся справа, специфицирует максимальную продолжительность вызова `getImage`.

Как правило, рекомендуется выбирать для сообщений короткие имена, чтобы не путать их с именами операций.

Моделирование распределения объектов

Моделирование распределения компонентов обсуждается в главе 15.

При моделировании топологии распределенной системы следует рассмотреть физическое расположение как узлов, так и артефактов. Если в центре внимания находится управление конфигурацией развернутой системы, то моделирование распределения узлов особенно важно для визуализации, специфицирования, конструирования и документирования размещения таких физических сущностей, как исполняемые модули, библиотеки и таблицы. Если же вас больше интересует функциональность, масштабируемость и пропускная способность системы, то важнее всего моделирование распределения объектов.

Моделирование процессов и потоков обсуждается в главе 23.

Принять решение о том, как распределить объекты в системе, сложно, и не только потому, что вопросы распределения объектов тесно связаны с вопросами параллелизма. Непродуманное решение может стать причиной очень низкой производительности, но слишком изощренные подходы немногим лучше, а пожалуй, даже и хуже, поскольку приводят к нестабильности.

Для моделирования распределения объектов следует воспользоваться следующими рекомендациями:

- Для каждого представляющего интерес класса объектов в системе рассмотреть местонахождение его ссылок – другими словами, выявить всех его соседей и их местоположение. Сильно связанное расположение означает, что логически соседние объекты находятся рядом, а слабо связанное – что они физически удалены друг от друга (и значит, при обмене информацией между ними будут иметь место временные задержки). Желательно размещать объекты рядом с действующими лицами, которые ими манипулируют.

- Рассмотреть образцы взаимодействия между взаимосвязанными наборами объектов. Расположить рядом наборы тесно взаимодействующих объектов, чтобы снизить затраты на коммуникацию. Разделить наборы слабо взаимодействующих объектов.
- Далее рассмотреть распределение обязанностей в системе. Перераспределить объекты так, чтобы сбалансировать загрузку каждого узла.
- Не забывать о безопасности, изменчивости и качестве услуг – учесть эти соображения при размещении объектов.
- Соотнести объекты с артефактами таким образом, чтобы тесно связанные объекты оказались в одном и том же артефакте.
- Соотнести артефакты с узлами таким образом, чтобы вычислительные потребности каждого узла оказались в пределах его возможностей. При необходимости добавить дополнительные узлы.
- Сбалансировать производительность и затраты на коммуникацию, размещая тесно связанные артефакты на одном узле.

Диаграммы объектов обсуждаются в главе 14.

На рис. 24.5 представлена диаграмма объектов, которая моделирует распределение объектов в системе розничной торговли. Ценность этой диаграммы в том, что она позволяет визуализировать физическое размещение ключевых объектов. Как видно, два объекта `Order` (Заказ) и `Sales` (Продажи) находятся в узле `Workstation` (РабочаяСтанция),

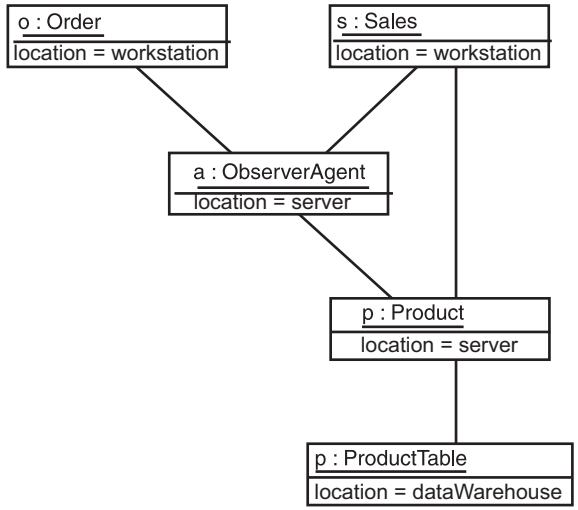


Рис. 24.5. Моделирование распределения объектов

два других (ObserverAgent – АгентНаблюдения и Product – Продукт) в узле Server и один (ProductTable – ТаблицаПродуктов) в узле DataWarehouse (ХранилищеДанных).

Советы и подсказки

Хорошо структурированная модель с пространственно-временными свойствами обладает следующими особенностями:

- описывает только те пространственно-временные свойства, которые необходимы и достаточны для понимания желаемого поведения системы;
- централизует использование этих свойств, так чтобы их было легко найти и модифицировать.

Изображая в UML пространственное или временное свойство, руководствуйтесь следующими принципами:

- давайте отметкам времени (то есть соответствующим сообщениям) осмысленные имена;
- проводите явное различие между временными выражениями, значениями которых является абсолютное и относительное время;
- показывайте пространственные свойства только тогда, когда важно визуализировать местонахождение элементов в развернутой системе;
- в более сложных случаях используйте профиль UML *«UML Profile for Schedulability, Performance, and Time»*. Данная спецификация OMG предназначена для высокопроизводительных систем реального времени.

Глава 25. Диаграммы состояний

Наряду с диаграммами состояний динамические аспекты поведения систем моделируют диаграммы последовательности, коммуникации, деятельности и вариантов использования. Диаграммы последовательности и коммуникации обсуждаются в главе 19, диаграммы деятельности – в главе 20, диаграммы вариантов использования – в главе 18.

В этой главе:

Диаграммы состояний – это один из пяти видов диаграмм UML, предназначенных для моделирования динамических аспектов поведения систем. Диаграмма состояний показывает конечный автомат. И диаграммы деятельности, и диаграммы состояний подходят для моделирования жизненного цикла объекта. Однако в то время, как диаграмма деятельности демонстрирует поток управления от одной деятельности к другой через множество объектов, диаграмма состояний отображает поток управления от состояния к состоянию внутри отдельного объекта.

Диаграммы состояния применяются для моделирования динамических аспектов поведения систем (в большинстве случаев это моделирование поведения реактивных объектов). *Реактивным* называется такой объект, поведение которого лучше всего характеризуется его реакцией на события, получаемые им извне его контекста. Реактивный объект имеет четкое время жизни, и его текущее поведение зависит от прошлого. Диаграммы состояний могут быть присоединены к классам, вариантам использования либо ко всей системе с целью визуализации, специфицирования, конструирования и документирования динамики отдельных объектов.

Диаграммы состояний полезны не только для моделирования динамических аспектов системы, но и для конструирования исполняемых систем посредством прямого и обратного проектирования.

Введение

Разница между построением собачьей будки и небоскреба обсуждается в главе 1.

Представим инвестора, который финансирует строительство небоскреба. Скорее всего, его не будут интересовать подробности процесса строительства. Выбор материала, планирование закупок, множество совещаний для обсуждения деталей – все это важно для строителей, но мало интересует того, кто финансирует проект.

Инвестор заинтересован в хорошем возврате своих инвестиций и в том, чтобы защитить их от риска. Очень доверчивый капиталовладелец дает застройщику кучу денег, затем надолго исчезает и возвращается только тогда, когда застройщик готов вручить ему ключи от здания. Такой инвестор в действительности заинтересован лишь в конечном результате строительства.

Более прагматичный предприниматель тоже в целом доверяет застройщику, однако, прежде чем расстаться с деньгами, предпочитает немного последить за работой. В этих целях осторожный инвестор устанавливает четкие этапы исполнения проекта, завершение каждого из которых будет означать выполнение определенного объема работ и послужит сигналом для финансирования следующего этапа. Например, вначале небольшая часть денег идет на оплату работы архитектора. После того как архитектурный проект утвержден, несколько большая часть денег отпускается на осуществление инженерных (расчетных) работ. После завершения данного этапа, если заинтересованные стороны удовлетворены результатом, еще большая часть денег выделяется для того, чтобы застройщик мог начать копать котлован.

Графики Гантта и Перта обсуждаются в главе 20.

Весь процесс строительства – от рытья котлована до получения сертификата владения домом – также разделен на этапы, каждый из которых фиксирует некоторое промежуточное состояние проекта: архитектурная часть, инженерная часть, земляные работы, подведение инфраструктуры, возведение стен и т.д. Для инвестора видеть состояние строительства важнее, чем следить за потоком деятельности; последнее имеет большее значение для строителя и может осуществляться с помощью графиков Перта, предназначенных для моделирования потока работ проекта.

Диаграммы деятельности, используемые в качестве блок-схем, обсуждаются в главе 20, автоматы – в главе 22.

Точно так же и при моделировании программных систем вы обнаружите, что наиболее естественный способ визуализировать, специфицировать, конструировать и документировать поведение некоторого рода объектов – сосредоточиться на потоке управления, ведущего от состояния к состоянию, а не от одной деятельности к другой. Второе обычно делается с помощью блок-схем (а в UML – с помощью диаграмм деятельности). Представим на минуту модель встроенной домашней системы сигнализации. Система работает непрерывно, реагируя на такие внешние события, как, например,

разбивание окна. Вдобавок пос ется описанием стабильных состояний (например, `Idle` – Простой, `Armed` – Готовность, `Active` – Активность, `Checking` – Проверка и т.п.), событий, инициирующих переход из одного состояния в другое, а также действий, выполняемых при каждом таком переходе.

В UML управляемое событиями поведение объекта моделируется с помощью диаграмм состояний. Как показано на рис. 25.1, диаграмма состояний – это просто представление автомата, подчеркивающее поток управления от одного состояния к другому.

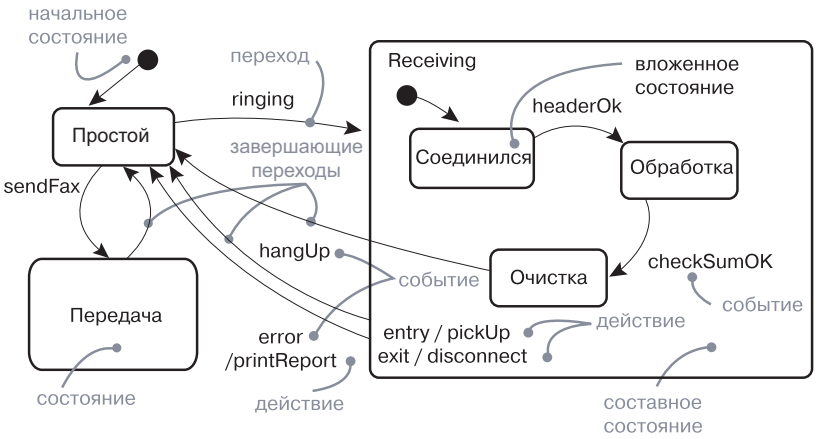


Рис. 25.1. Диаграмма состояний

Базовые понятия

Диаграмма состояний (state diagram) показывает автомат, сосредотачивая внимание на потоке управления от одного состояния к другому. Изображается в виде графа с вершинами и дугами (ребрами).

Автомат (state machine) – это описание последовательности состояний, через которые проходит объект на протяжении жизненного цикла, реагируя на события, а также описание реакции на эти события.

Состояние (state) – ситуация в жизненном цикле объекта, на протяжении которой он удовлетворяет некоторому условию, выполняет некоторую деятельность или ожидает некоторого события.

Событие (event) – спецификация существенного факта, который происходит во времени и пространстве. В контексте автомата событие – это воздействие, которое вызывает переход между состояниями.

Переход (transition) – связь между двумя состояниями, показывающая, что объект, находящийся в первом состоянии, должен выполнить некоторые действия и перейти во второе, как только

произойдет определенное событие и будут выполнены определенные условия.

Деятельность (activity) специфицирует работу, происходящую внутри автомата.

Действие (action) – примитивное выполняемое вычисление, приводящее к смене состояния модели или возврату значения.

На заметку. Диаграммы состояний, используемые в UML, основаны на нотации, предложенной Дэвидом Харелом (David Harel). В частности, концепции вложенных и ортогональных состояний были подробно разработаны Харелом в виде формальной системы. В UML эти концепции менее формализованны, чем в нотации Харела, и отличаются некоторыми деталями – в частности, сосредотачивая внимание на объектно-ориентированных системах.

Общие свойства

Общие свойства диаграмм обсуждаются в главе 7.

Диаграмма состояний обладает свойствами, общими для всех диаграмм: имеет имя и графическое содержимое, представляющее собой проекцию модели. От прочих диаграмм она отличается именно своим содержимым.

Содержимое

Простые и составные состояния, переходы, события и действия обсуждаются в главе 22, диаграммы деятельности – в главе 20, примечания и ограничения – в главе 6.

Диаграммы состояния обычно содержат простые и составные состояния, а также переходы, события и действия. Кроме того, как и все диаграммы, могут включать примечания и ограничения.

На заметку. Диаграмма состояний, по сути, состоит из элементов, встречающихся в любом автомате. Она может содержать ветвления, разделения, соединения, состояния действий и состояния деятельности, объекты, начальные и конечные состояния, исторические состояния и т.д. – в общем, на диаграмме состояний отображаются все без исключения характеристики автомата.

Общее применение

Диаграммы состояний применяются для моделирования динамических аспектов поведения системы. Имеется в виду обусловленное порядком возникновения событий поведение объектов любого рода в любом представлении системной архитектуры, включая классы (в т.ч. активные), интерфейсы, компоненты и узлы.

Пять представлений архитектуры обсуждаются в главе 2, экземпляры – в главе 13, классы – в главах 4 и 9.

Активные классы обсуждаются в главе 23, интерфейсы – в главе 11, компоненты – в главе 15, узлы – в главе 27, – в главе 17, системы – в главе 32.

Диаграммы состояний можно использовать для моделирования некоторого динамического аспекта системы, притом в контексте почти любого элемента модели. Однако чаще их применяют в контексте всей системы, подсистемы либо класса. Также диаграммы состояний можно присоединять к вариантам использования (для моделирования сценариев).

При моделировании динамических аспектов системы, класса или варианта использования диаграммы состояний обычно применяются для моделирования реактивных объектов.

Реактивный (reactive), или *управляемый событиями*, объект – такой, поведение которого лучше всего характеризуется его реакцией на события, принимаемые им извне его контекста. Обычно реактивный объект простаивает в ожидании события. Когда он получает это событие, его реакция обычно зависит от предыдущих событий. Отреагировав определенным образом, объект опять возвращается в состояние простоя, ожидая следующего события. Рассматривая объект подобного рода, вы сосредотачиваете внимание на его стабильных состояниях, событиях, инициирующих переходы от одного состояния к другому, а также действиях, которые выполняются при каждой смене состояния.

На заметку. В отличие от диаграмм состояний, диаграммы действий применяются для моделирования потока работ или моделирования операций. Диаграммы деятельности лучше подходят для моделирования потока деятельности во времени, как это представляется на блок-схемах.

Типичные приемы моделирования

Моделирование реактивных объектов

Взаимодействия обсуждаются в главе 16, диаграммы деятельности – в главе 20.

Чаще всего, как уже было сказано, диаграммы состояний используются при моделировании поведения реактивных объектов, особенно экземпляров классов, вариантов использования и системы в целом. В то время как взаимодействия моделируют поведение сообщества объектов, работающих вместе, диаграмма состояний моделирует поведение отдельного объекта во время его жизненного цикла. В отличие от диаграмм деятельности, которые моделируют поток управления от одной деятельности к другой, диаграмма состояний моделирует поток управления от одного события к другому.

Моделирование жизненного цикла объектов обсуждается в главе 22.

При моделировании поведения реактивного объекта, по сути, специфицируются три вещи: стабильные состояния, в которых объект может пребывать, события, вызывающие переходы от состояния к состоянию, и действия, выполняемые при каждой смене состояний. Кроме того, моделирование поведения реактивного объекта подразумевает моделирование его жизненного цикла начиная с создания объекта вплоть до его уничтожения, с выделением стабильных состояний, в которых он может находиться.

Время и пространство обсуждаются в главе 24.

Стабильное состояние представляет условие, при котором объект может существовать в течение некоторого определенного периода времени. Когда происходит событие, объект может переходить из одного состояния в другое. Эти события также могут вызывать переходы в себя и внутренние переходы, когда исходное и целевое состояния объекта совпадают. Реагируя на события и изменения состояния, объект может выполнять действия.

На заметку. Когда моделируется поведение реактивного объекта, можно специфицировать его действия, привязывая их к переходу или смене состояния. В технической терминологии автомат, все действия которого привязаны к переходам, называется *машиной Мили* (Mealy machine), а автомат, действия которого привязаны к состояниям, – *машиной Мура* (Moore machine). В математическом плане эти два варианта эквивалентны по мощности. На практике чаще создаются диаграммы состояний, использующие некоторую комбинацию машин Мили и Мура.

Пред- и постусловия обсуждаются в главе 9, интерфейсы – в главе 11.

Чтобы смоделировать реактивный объект, необходимо:

- Выбрать контекст автомата – класс, вариант использования или систему в целом.
- Установить начальное и конечное состояния объекта. Возможно, чтобы управлять остальными частями модели, понадобится также установить пред- и постусловия начального и конечного состояний соответственно.
- Принять решения относительно стабильных состояний объекта, рассмотрев условия, в которых объект может существовать в течение определенного периода времени. Начать с состояний высшего уровня и только после этого перейти к рассмотрению возможных подсостояний.
- Принять решения относительно осмысленного частичного упорядочения состояний на протяжении времени жизни объекта.

- Принять решения относительно событий, которые могут вызывать переходы из одного состояния в другое. Смоделировать эти события как триггеры переходов между состояниями.
- Присоединить действия к переходам (как в машине Мили) и/или к состояниям (как в машине Мура).
- Рассмотреть возможности упрощения автомата за счет применения подсостояний, ветвления, разделений, соединений и исторических состояний.
- Убедиться, что все состояния достижимы при некотором стечении событий.
- Проверить, нет ли каких-нибудь «мертвых» состояний, из которых объект не сможет выйти ни при каком сочетании событий.
- Провести трассировку автомата (либо вручную, либо с применением инструментальных средств), чтобы подтвердить соответствие ожидаемым последовательностям событий и реакциям на них.

В качестве примера на рис. 25.2 показана диаграмма состояний, предназначенная для разбора некоего простого контекстно-свободного языка – вроде того, что можно встретить в системах, обрабатывающих входные и выходные потоки сообщений XML.

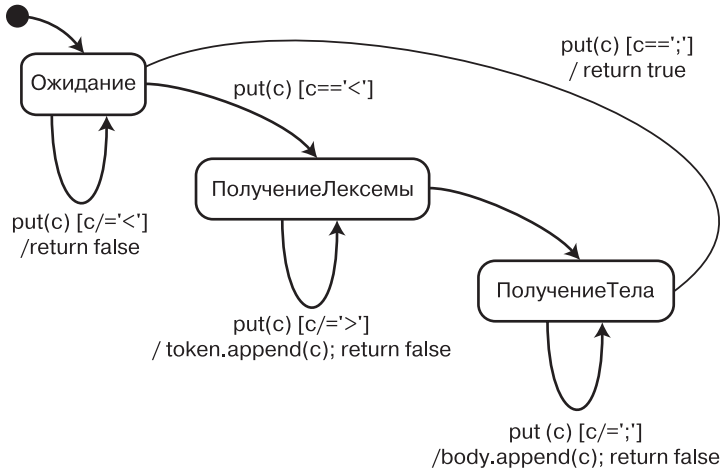


Рис. 25.2. Моделирование реактивных объектов

В данном случае автомат предназначен для разбора потока символов, соответствующих синтаксису

```
message : '<' string '>' string ';'
```

Первая строка представляет тег, вторая – тело сообщения. Из полученного потока символов принимаются только сообщения с правильным синтаксисом.

События обсуждаются в главе 21.

Как показано на рисунке, у автомата есть только три стабильных состояния: Waiting (Ожидание), GettingToken (ПолучениеСлова) и GettingBody (ПолучениеТела). Состояние спроектировано в виде машины Мили – с действиями, присоединенными к переходам. Фактически есть только одно событие, которое может нас интересовать при рассмотрении этого автомата – вызов put с параметром c (символом). В состоянии Waiting автомат отбрасывает любой символ, который не означает начало слова (что объявлено в защитном условии). Как только встречается начало слова, состояние объекта изменяется на GettingToken. Находясь в этом состоянии, автомат сохраняет каждый символ, который не является окончанием слова. Когда приходит символ окончания слова, состояние объекта изменяется на GettingBody. Находясь в этом состоянии, автомат сохраняет каждый символ, который не означает окончания тела сообщения (что объявлено в защитном условии). Как только поступает окончание сообщения, состояние объекта изменяется на Waiting и возвращается значение, указывающее на то, что сообщение было разобрано и автомат готов получать следующее.

Отметим, что последнее состояние специфицирует автомат, который работает непрерывно, – конечного состояния не существует.

Прямое и обратное проектирование

Прямое проектирование (создание кода из модели) возможно для диаграммы состояний, особенно если контекстом таковой является класс. Например, используя диаграмму состояний, показанную на рис. 25.2, инструмент прямого проектирования может сгенерировать следующий Java-код для класса MessageParser (АнализаторСообщений):

```
class MessageParser {
public
 boolean put(char c) {
 switch (state) {
  case Waiting:
  if (c == '<') {
   state = GettingToken;
   token = new StringBuffer();
   body = new StringBuffer();
  }
  break;
```



```
  case GettingToken :
  if (c == '>')
   state = GettingBody;
  else
   token.append(c);
  break;
  case GettingBody :
  if (c == ';')
   state = Waiting;
  else
   body.append(c);
  return true;
 }
 return false;
 }
 StringBuffer getToken() {
 return token;
 }
 StringBuffer getBody() {
 return body;
 }
private
 final static int Waiting = 0;
 final static int GettingToken = 1;
 final static int GettingBody = 2;
 int state = Waiting;
 StringBuffer token, body;
}
```

Анализ такого кода требует небольшого напряжения ума. Инструмент прямого проектирования должен генерировать необходимые закрытые атрибуты и конечные статические константы.

Обратное проектирование (создание модели из кода) теоретически возможно, но практически не очень полезно. Выбор того, что составляет осмысленное состояние объекта, зависит от взгляда проектировщика. Инструменты обратного проектирования не обладают способностью к абстрагированию, а потому не могут генерировать осмысленные диаграммы состояний. Более интересной, чем обратное проектирование модели из кода, может оказаться анимация модели на основе работы реальной системы. Например, если взять диаграмму с рис. 25.2, инструментальное средство могло бы анимировать состояния автомата на ней по мере того, как их принимала бы работающая система. Аналогичным образом можно анимировать выполнение переходов, показывая момент получения события и результат выполнения действия. Под управлением

отладчика вы могли бы контролировать скорость выполнения, устанавливать точки прерывания для приостановки работы в интересные моменты времени, чтобы проверить значения атрибутов отдельных объектов.

Советы и подсказки

Создавая диаграмму состояний в UML, следует помнить, что она представляет собой всего лишь проекцию одной модели динамических аспектов системы. Отдельно взятая диаграмма состояний может охватить семантику одного реактивного объекта, но семантику непростой системы не в состоянии передать.

Хорошо структурированная диаграмма состояний обладает следующими характеристиками:

- сосредоточена на передаче одного аспекта динамики системы;
- содержит только те элементы, которые существенны для понимания данного аспекта;
- детализирована адекватно своему уровню абстракции (раскрывает только те сущности, которые важны для понимания);
- применяет сбалансированное сочетание машин Мили и Мура.

Когда вы рисуете диаграмму состояний, пользуйтесь следующими рекомендациями:

- присваивайте диаграмме имя, отражающее ее назначение;
- начинайте с моделирования стабильных состояний объекта, затем моделируйте допустимые переходы между состояниями. Показывайте ветвление, параллелизм и потоки объектов во вторую очередь – возможно, на отдельных диаграммах;
- располагайте элементы так, чтобы свести к минимуму пересечение линий;
- при создании объемных диаграмм состояний рассмотрите возможность применения расширенных средств – таких как вложенные автоматы, которые включены в полную спецификацию UML.

Часть VI

Моделирование архитектуры

Глава 26. Артефакты

В этой главе:

- Артефакты, классы и их материализация
- Моделирование исполняемых программ и библиотек
- Моделирование таблиц, файлов и документов
- Моделирование исходного кода

Артефакты живут в материальном мире битов и потому являются важнейшими строительными блоками при моделировании физических аспектов системы. Артефакт – это физическая и заменяемая часть системы.

Артефакты используются для моделирования таких физических сущностей, которые могут располагаться на узле, – например, исполняемых программ, библиотек, таблиц, файлов и документов. Обычно артефакт представляет собой физическую группировку таких логических элементов, как классы, интерфейсы и кооперации.

Введение

Конечный результат работы строительной компании – здание в его материальном воплощении. Для визуализации, специфицирования и документирования устройства жилого дома (расположения стен, дверей и окон, размещения систем электроснабжения и водопровода и т.д.), а также общего архитектурного стиля вы строите логические модели. Во время строительства стены, двери, окна и другие концептуальные сущности, данные вам в абстракциях, превращаются в реальные, физические объекты.

Разница между строительством собачьей будки и небоскреба обсуждается в главе 1.

Все эти предварительные представления крайне необходимы. Если вы строите здание, стоимость разрушения и переделки которого практически равна нулю (например, собачью будку), то, вероятно, можете сразу приступить к физическому строительству, миновав этап логического моделирования. Если же речь идет о чем-то долговечном, то стоимость ошибок и переделок высока; в этом случае построение логической и физической моделей – разумный способ снизить степень риска.

Аналогичным образом обстоит дело и с программными системами. Вы осуществляете логическое моделирование для визуализации, специфицирования и документирования решений относительно словаря предметной области, а также структурных и поведенческих способов кооперации сущностей. Физическое же моделирование требуется для построения исполняемой системы. Если логические сущности «живут» в концептуальном мире, то физические пребывают в мире битов, то есть в конечном счете располагаются на физических узлах и могут быть исполнены непосредственно либо каким-то косвенным образом участвовать в работе исполняемой системы.

В UML все физические сущности моделируются как артефакты. Артефакт – это физическая сущность на уровне платформы реализации.

Многие операционные системы и языки программирования непосредственно поддерживают концепцию артефакта. Объектные библиотеки, исполняемые программы, компоненты .NETи Enterprise Java Beans, – все это примеры артефактов, которые могут быть представлены на UML. Но артефакты могут быть использованы и для представления других сущностей, которые принимают участие в работе исполняемых программ, – таких, как таблицы, файлы и документы.

Стереотипы обсуждаются в главе 6.

В UML артефакты изображаются так, как показано на рис. 26.1. Эта каноническая нотация позволяет визуализировать артефакт независимо от операционной системы и языка программирования. Используя стереотипы – один из механизмов расширения UML, можно приспособить эту нотацию для представления специфических разновидностей артефактов.

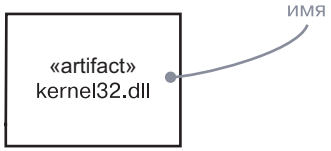


Рис. 26.1. Артефакты

Базовые понятия

Артефакт – это физическая часть системы, которая существует на уровне платформы реализации. Изображается в форме прямоугольника с ключевым словом «artifact» внутри.

Имена

Каждый артефакт должен обладать *именем*, отличающим его от других артефактов. Имя представляет собой текстовую строку; взятая в отдельности, она называется *простым именем*. Имя,

Имя артефакта должно быть уникальным в пределах включающего его узла.

предваренное именем пакета, в котором находится данный артефакт, называется *квалифицированным*. Обычно, изображая артефакт, указывают только его имя. Как и в случае с классами, вы можете дополнять пиктограммы артефактов помеченными значениями или дополнительными разделами, чтобы уточнить подробности (см. рис. 26.2).

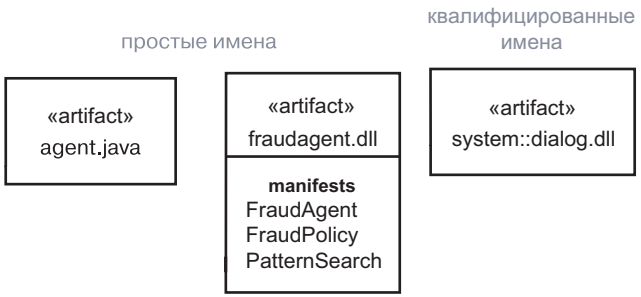


Рис. 26.2. Простые и квалифицированные имена артефактов

На заметку. Имя артефакта представляет собой текст, содержащий в любом количестве буквы латинского алфавита, цифры и ряд знаков препинания (за исключением таких, как двоеточия, которые применяются для отделения имени артефакта от имени включающего его пакета). Имя может записываться в несколько строк. На практике в качестве имен артефактов выступают существительные, взятые из словаря реализации и, в зависимости от целевой операционной системы, включающие расширения (такие как java или dll).

Артефакты и классы

Классы обсуждаются в главах 4 и 9, взаимодействия – в главе 16.

И классы, и артефакты являются классификаторами. Однако между ними есть существенная разница:

- классы представляют логические абстракции, а артефакты – физические сущности, которые существуют в мире битов. В силу этого артефакты могут располагаться на узлах, а классы – нет;
- артефакты представляют физическую упаковку битов на платформе реализации;
- классы могут иметь атрибуты и операции. Артефакты могут реализовывать классы и методы, но сами по себе не имеют атрибутов и операций.

Первое отличие наиболее важно. Когда моделируется система, решение о том, нужно ли использовать классы или артефакты, сводится к простому выбору: если моделируемая вами сущность «живет» непосредственно на узле, применяйте артефакт; в противном случае используйте класс. Подтверждением тому служит и второе отличие.

Третье отличие предполагает связь между классами и артефактами. В частности, артефакт – это физическая реализация набора логических элементов, таких как классы и кооперации. Как показано на рис. 26.3, связь между артефактом и классами, которые он реализует, может быть показана явно с помощью материализации.

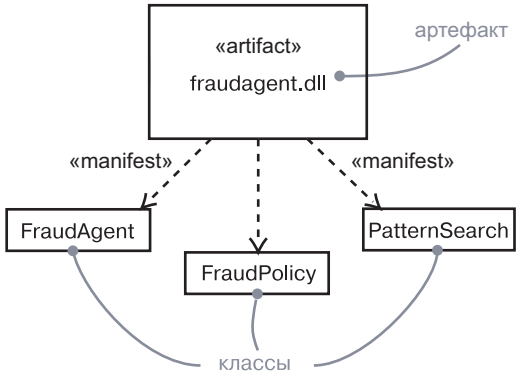


Рис. 26.3. Артефакты и классы

Виды артефактов

Можно выделить три вида артефактов:

1. *Артефакты размещения* (deployment artifacts) необходимы и достаточны для формирования исполняемой системы. Примеры – динамические библиотеки (DLL) и исполняемые программы (EXE). Определение артефакта UML достаточно широко, чтобы вместить классические объектные модели, такие как .NET, CORBA и Enterprise Java Beans, а равно и альтернативные объектные модели (возможно, включающие динамические Web-страницы, таблицы баз данных, исполняемые программы, использующие собственные коммуникационные механизмы);
2. *Артефакты рабочих продуктов* (work product artifacts), по существу, представляют собой результаты процесса разработки. Примеры – файлы исходного кода и файлы данных, из которых создаются артефакты размещения. Артефакты такого рода напрямую не участвуют в исполняемой системе, но являются продуктами разработки, используемыми для ее создания;

3. *Артефакты исполнения* (execution artifacts) создаются в результате работы системы. Примеры – объект .NET, который создается из DLL.

Стандартные элементы

Механизмы расширения UML обсуждаются в главе 6.

Механизмы расширения UML применимы к артефактам. Чаще всего для расширения свойств артефактов (таких, например, как спецификация версии артефакта разработки) используются помеченные значения, а для определения новых видов артефактов (например, специфичных для конкретной операционной системы) – стереотипы.

Перечислим предопределенные стандартные стереотипы UML, применимые к артефактам:

1. `Executable` – специфицирует артефакт, который может быть выполнен на узле;
2. `Library` – специфицирует статическую или динамическую библиотеку объектов;
3. `File` – специфицирует артефакт, который представляет документ, содержащий исходный код или данные;
4. `Document` – специфицирует артефакт, представляющий документ.

Другие стереотипы могут быть определены для конкретных платформ и систем.

Типичные приемы моделирования

Моделирование исполняемых программ и библиотек

Чаще всего цель применения артефактов заключается в моделировании артефактов размещения, составляющих реализацию системы. Если вы устанавливаете простую систему, реализация которой состоит всего лишь из одного исполняемого файла, нет необходимости ни в каком моделировании артефактов. Если же система состоит из нескольких исполняемых программ и ассоциированных с ними объектных библиотек, моделирование артефактов помогает визуализировать, специфицировать, конструировать и документировать принятые решения относительно ее физической организации. Моделирование артефактов еще более важно, если требуется контролировать версии и управлять конфигурацией частей системы.

На подобные решения влияет и топология целевой системы (см. главу 27).

Для большинства систем артефакты размещения основываются на решениях относительно того, как должна быть сегментирована физическая реализация системы. На такие решения влияет множество соображений технического характера (в частности, выбранные средства операционной системы), удобства управления конфигурацией (например, вероятность изменения некоторых частей системы по прошествии времени), а также неоднократного использования (повторное применение ряда артефактов другими системами).

Чтобы смоделировать исполняемые программы или библиотеки, необходимо:

- Идентифицировать разбиение физической системы на части. Рассмотреть влияние технических требований, требований управления конфигурацией и повторного использования.
- Смоделировать любые исполняемые программы и библиотеки как артефакты, используя соответствующие стандартные элементы. Если ваша реализация представляет новые виды артефактов, создайте специальные стереотипы.
- Обращать внимание на соединения частей системы, моделируя интерфейсы, которые одни артефакты используют, а другие реализуют.
- Смоделировать связи между исполняемыми программами, библиотеками и интерфейсами для реализации вашего замысла. Чаще всего понадобится моделировать зависимости между этими частями, чтобы визуализировать воздействие изменений.

В качестве примера на рис. 26.4 показан набор артефактов, составляющих персональный инструмент измерения производительности, который запускается на отдельном компьютере. Здесь изображены одна исполняемая программа (`animator.exe`) и четыре библиотеки (`dlog.dll`, `wframe.dll`, `render.dll` и `raytrce.dll`); все они используют стандартные элементы UML, описывающие, соответственно, исполняемые программы библиотеки. Кроме прочего, данная диаграмма представляет зависимости между артефактами.

Пакеты обсуждаются в главе 12.

По мере роста ваших моделей вы обнаружите, что многие артефакты, концептуально и семантически близкие, имеют тенденцию собираться в группы. Для моделирования таких кластеров артефактов в UML можно использовать пакеты.

Моделирование размещения обсуждается в главе 27.

Для крупных систем, которые размещаются на нескольких компьютерах, может понадобиться моделирование способа распределения артефактов с указанием узлов, на которых они находятся.

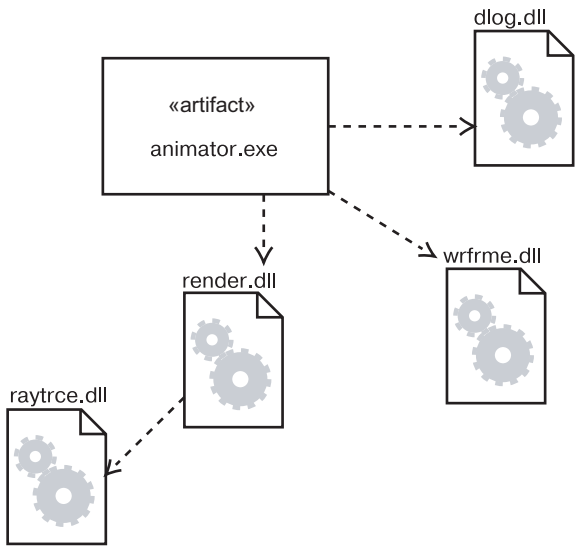


Рис. 26.4. Моделирование исполняемых программ и библиотек

Моделирование таблиц, файлов и документов

Моделирование исполняемых программ и библиотек, составляющих физическую реализацию системы, полезно, но часто обнаруживается, что существует множество вспомогательных артефактов, которые не являются ни исполняемыми файлами, ни библиотеками, хотя не менее важны для физического размещения системы. Например, реализация может включать файлы данных, вспомогательную документацию, скрипты, файлы протоколов, файлы инициализации, а также файлы, используемые при установке/удалении системы. Моделирование подобных артефактов – важная часть управления конфигурацией системы. К счастью, для этого хорошо подходят артефакты UML.

Чтобы смоделировать таблицы, файлы и документы, необходимо:

- Идентифицировать вспомогательные артефакты, являющиеся частью физической реализации системы.
- Смоделировать эти сущности в виде артефактов. Если ваша реализация представляет новые виды артефактов, понадобится ввести соответствующие стереотипы.
- При необходимости обозначить связи между этими вспомогательными артефактами и другими исполняемыми программами,

библиотеками и интерфейсами системы. Чаще всего оказывается важным моделирование зависимостей между этими частями, позволяющее визуализировать влияние изменений.

Логическое и физическое моделирование баз данных обсуждается в главах 8 и 30 соответственно.

Обратимся к рис. 26.5, который базируется на рис. 26.4. Здесь показаны таблицы, файлы и документы, являющиеся частями установленной системы, окружающими исполняемый файл `animator.exe`. Имеются в наличии один документ (`animator.hlp`), один простой файл (`animator.ini`) и одна таблица базы данных (`shapes.tbl`). Также данный пример иллюстрирует некоторые пользовательские стереотипы и пиктограммы артефактов.

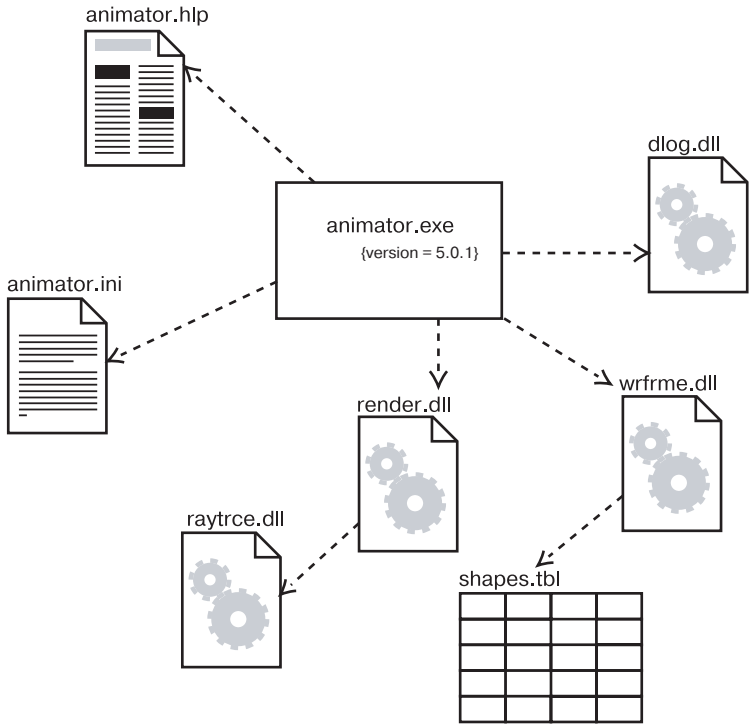


Рис. 26.5. Моделирование таблиц, файлов и

Моделирование баз данных значительно усложняется, когда вы начинаете иметь дело со множеством таблиц, триггеров и хранимых процедур. Чтобы визуализировать, специфицировать, конструировать и документировать эти средства, вам придется смоделировать логическую схему наряду с физической базой данных.

Моделирование исходного кода

Главное назначение артефактов – моделирование физических частей реализации. Второе по важности их применение – моделирование конфигурации всех файлов исходного кода, которые нужны средству разработки для построения исполняемых артефактов. Эти файлы представляют собой артефакты продуктов процесса разработки.

Моделирование исходного кода в графическом виде полезно, в частности, для визуализации зависимостей компиляции между файлами исходного кода, а также для управления расщеплением и слиянием групп этих файлов при распараллеливании и соединении самого процесса разработки. В этом смысле артефакты UML могут быть графическим интерфейсом инструментов управления конфигурацией и контроля версий.

Для большинства систем файлы исходного кода создаются с учетом решений, принятых вами относительно того, как должен быть сегментирован исходный код для нужд среды разработки. Эти файлы используются для хранения деталей реализации классов, интерфейсов, кооперации и других логических элементов – промежуточной фазы процесса создания физических бинарных артефактов, которые производятся из таких элементов инструментальными средствами разработки. Большей частью подобные инструменты определяют стиль организации (часто – один или два файла на класс), но все же для разработчика полезно визуализировать связи между этими файлами. Решения относительно того, как организовать исходные файлы в пакеты и как управлять их версиями, принимаются на основе выбранных способов управления изменениями.

Чтобы смоделировать исходный код, необходимо:

- С учетом ограничений, накладываемых средствами разработки, смоделировать файлы, используемые для хранения деталей реализации логических элементов, наряду с их зависимостями компиляции.
- Если важно, чтобы модели были согласованы с инструментами разработки и управления конфигурацией, для каждого файла, подверженного управлению конфигурацией, включить помеченные значения – такие как информация о версии, авторе и помещении файла в репозиторий / извлечении из репозитория.
- По возможности позволять инструментам разработки управлять связями между этими файлами и использовать UML только для того, чтобы визуализировать и документировать такие связи.

На рис. 26.6 показаны некоторые файлы исходного кода, используемые для построения библиотеки `render.dll` из предыдущих примеров. Рисунок включает четыре заголовочных файла (`render.h`, `rengine.h`, `poly.h` и `colortab.h`), которые представляют исходный код спецификации определенных классов. Также имеется файл реализации (`render.cpp`), представляющий реализацию одного из этих заголовков.

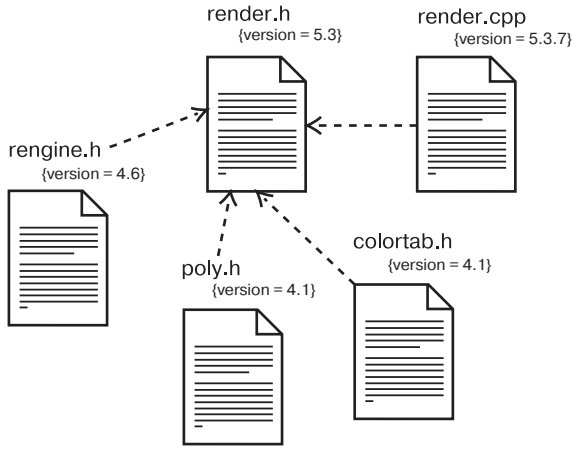


Рис. 26.6. Моделирование исходного кода

Пакеты обсуждаются в главе 12. Связь трассировки как разновидность зависимости рассматривается в главах 5 и 10.

По мере развития ваших моделей вы обнаружите, что многие файлы исходного кода, концептуально и семантически связанные, имеют тенденцию собираться в группы. Как правило, инструменты разработки будут размещать их в разных директориях. В UML для моделирования подобных кластеров исходного кода можно использовать пакеты.

Кроме того, в UML есть возможность визуализировать отношения между классом и его файлом исходного кода, а также, в свою очередь, связь файла исходного кода с исполняемой программой или библиотекой посредством трассировки (trace relationship). Однако необходимость в такой детализации при моделировании возникает редко.

Советы и подсказки

Работая с артефактами в UML, важно помнить о том, что вы осуществляете моделирование в физическом измерении. Хорошо структурированный артефакт обладает следующими характеристиками:

- непосредственно реализует набор классов, работающих совместно для обслуживания определенной семантики интерфейсов, – экономно и элегантно;
- слабо связан с другими артефактами.

Глава 27. Размещение

В этой главе:

Узлы, равно как и артефакты, существуют в материальном мире и являются важными строительными блоками при моделировании физических аспектов системы. Узел – это физический элемент, который существует во время выполнения и представляет вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а зачастую также и процессором.

Узлы используются для моделирования топологии аппаратных средств, на которых работает система. Как правило, узел – это процессор или устройство, на котором могут быть размещены артефакты.

Хорошо спроектированные узлы точно соответствуют словарю аппаратного обеспечения области решения.

Введение

Моделирование непрограммных сущностей обсуждается в главе 4, пять видов системной архитектуры описаны в главе 2.

Артефакты, которые вы разрабатываете или повторно используете в программной системе, должны быть размещены на какой-то аппаратуре, иначе они не смогут выполняться. Собственно, программная система и состоит из этих двух частей: программного и аппаратного обеспечения.

При проектировании архитектуры программной системы приходится рассматривать как логические, так и физические аспекты. К логическим элементам относятся такие сущности, как классы, интерфейсы, кооперации, взаимодействия и автоматы, а к физическим – артефакты (представляющие физическую упаковку логических сущностей) и узлы (представляющие аппаратуру, на которой размещаются и выполняются артефакты).

Стереотипы обсуждаются в главе 6.

Графическое изображение узла в UML показано на рис. 27.1. Это каноническое обозначение позволяет визуализировать узел, не конкретизируя стоящей за ним аппаратуры. С помощью стереотипов – одного из механизмов расширения UML – можно адаптировать эту нотацию для представления конкретных процессоров и устройств.

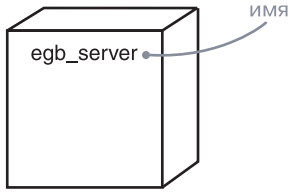


Рис. 27.1. Узлы

На заметку. Язык UML предназначен главным образом для моделирования программных систем, хотя в сочетании с языками моделирования аппаратных средств типа VHDL он может оказаться весьма полезным и при моделировании аппаратных систем. Кроме того, UML обладает достаточной выразительной мощностью для моделирования топологии автономных (stand-alone), встроенных (embedded), клиент-серверных (client-server) и распределенных (distributed) систем.

Базовые понятия

Узел (Node) – это физический элемент, который существует во время выполнения и представляет вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а зачастую и процессором. Изображается в виде куба.

Имена

Имя узла должно быть уникальным внутри включающего его пакета, это обсуждается в главе 12.

У каждого узла должно быть имя, отличающее его от прочих узлов. Имя представляет собой текстовую строку. Взятое само по себе, оно называется *простым именем*. К *квалифицированному имени* спереди добавлено имя пакета, в котором находится данный узел.

Обычно при изображении узла указывают только его имя, как видно из рис. 27.2. Но, как и в случае с классами, вы можете снабжать узлы помеченными значениями или дополнительными разделами, чтобы акцентировать детали.

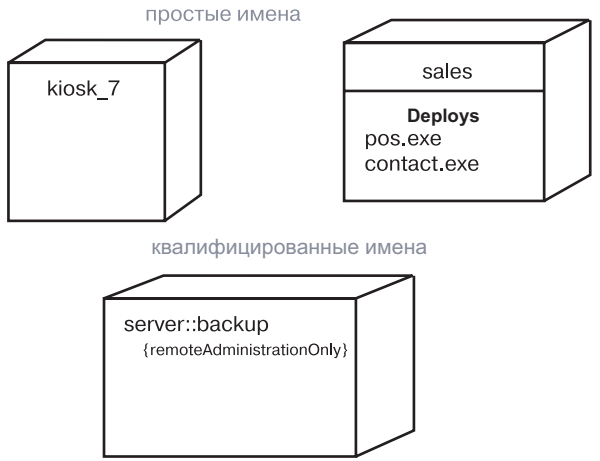


Рис. 27.2. Узлы с простыми и квалифицированными именами

На заметку. Имя узла может состоять из любых букв латинского алфавита, цифр и некоторых знаков препинания (за исключением таких, как двоеточие, которое применяется для отделения имени узла от имени включающего пакета). Имя может занимать несколько строк (количество символов не ограничено). На практике для именования узлов используют одно или несколько существительных, взятых из словаря реализации.

Узлы и артефакты

Артефакты обсуждаются в главе 26.

Во многих отношениях узлы подобны артефактам. Те и другие наделены именами, могут участвовать в связях зависимости, обобщения и ассоциации, бывают вложенными, могут иметь экземпляры и вступать во взаимодействия. Однако между ними есть и существенные различия:

- артефакты принимают участие в работе системы, а узлы – это сущности, на которых работают артефакты;
- артефакты представляют физическую упаковку логических элементов, узлы представляют физическое размещение артефактов.

Первое из этих отличий самое важное. Суть его проста: узлы исполняют артефакты, артефакты работают на узлах.

Зависимости обсуждаются в главе 5 и 10.

Второе различие предполагает наличие некоторой связи между классами, артефактами и узлами. В самом деле, артефакт – это материализация множества логических элементов, таких как классы и кооперации, а узел – место, на котором размещены артефакты. Класс может быть реализован одним или несколькими артефактами, а артефакт, в свою очередь, размещен в одном или нескольких узлах. Как показано на рис. 27.3, связь между узлом и артефактами, которые на нем размещены, может быть явно изображена с помощью вложенности. Как правило, вам не придется визуализировать такие связи. Лучше обозначать их как часть спецификации узла, используя, например, таблицу.

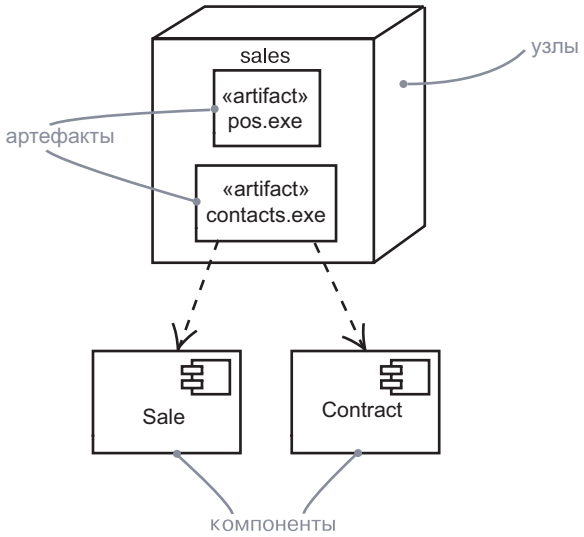


Рис. 27.3. Узлы и артефакты

Множество объектов или артефактов, приписанных к узлу как группа, называется *элементом распределения* (distribution unit).

На заметку. Узлы подобны классам в том отношении, что для них можно задать атрибуты и операции. Например, можно указать, что у узла есть атрибуты `скоростьПроцессора` и `память`, а также операции `включить`, `выключить`, `приостановить`.

Организация узлов

Пакеты обсуждаются в главе 12.

Узлы можно группировать в пакеты точно так же, как классы и артефакты.

Можно также организовывать узлы, специфицируя связи зависимости, обобщения и ассоциации (включая агрегации) между ними.

Связи обсуждаются в главе 5 и 10.

Соединения

Самый распространенный вид связи между узлами – это *ассоциация*. В данном контексте ассоциация представляет физическое соединение узлов, например линию Ethernet, последовательный канал или разделяемую шину (см. рис. 27.4). Ассоциации можно использовать даже для моделирования непрямых соединений типа спутниковой линии связи между двумя удаленными процессорами.

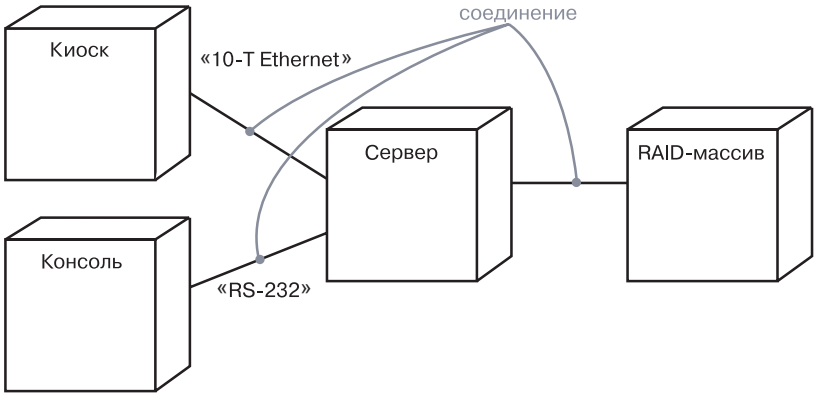


Рис. 27.4. Соединения

Поскольку узлы аналогичны классам, в нашем распоряжении находится весь аппарат ассоциаций. Иными словами, можно использовать роли, множественность и ограничения. Как показано на рис. 27.4, следует применять стереотипы, если необходимо моделировать разные виды соединений – к примеру, чтобы отличить соединение 10-T Ethernet от соединения по последовательному каналу RS-232.

Типичные приемы моделирования

Моделирование процессоров и устройств

Механизмы расширения UML обсуждаются в главе 6.

Моделирование процессоров и устройств, образующих топологию автономной, встроенной, клиент-серверной или распределенной системы, – вот самый распространенный пример использования узлов.

Поскольку все механизмы расширения UML применимы и к узлам, то для описания новых видов узлов, представляющих конкретные процессоры и устройства, часто используются стереотипы. *Процессор* (processor) – это узел, способный обрабатывать данные, то есть исполнять артефакт. *Устройство* (device) – это узел, не способный обрабатывать данные (по крайней мере, на выбранном уровне абстракции) и в общем случае используемый для представления чего-либо связанного с реальным миром.

Для моделирования процессоров и устройств вам понадобится:

- Идентифицировать вычислительные элементы представления системы с точки зрения размещения и смоделировать каждый из них как узел.
- Если эти элементы представляют процессоры и устройства общего вида, приписать им соответствующие стандартные стереотипы. Если же это процессоры и устройства, входящие в словарь предметной области, – сопоставить им подходящие стереотипы с пиктограммой.
- Как и в случае моделирования классов, рассмотреть атрибуты и операции, применимые к каждому узлу.

В качестве примера на рис. 27.5 изображена диаграмма, которую мы рассматривали выше. Теперь каждому узлу на ней приписан стереотип. Здесь `server` (сервер) – это узел со стереотипом процессора общего вида; `kiosk` (киоск) и `console` (консоль) – узлы со стереотипами специализированных процессоров, а `RAID farm` (RAID-массив) – узел со стереотипом специализированного устройства.

На заметку. Узлы – это, возможно, именно те строительные блоки UML, которым стереотипы приписываются чаще всего. Когда в ходе проектирования программной системы вы моделируете ее с точки зрения размещения, очень важно представлять потенциальным читателям визуальную информацию. Моделируя процессор, являющийся компьютером общего назначения, присвойте ему пиктограмму компьютера. Моделируя какое-либо устройство, например сотовый телефон, факс, модем или видеокамеру, обозначьте и его подходящей пиктограммой.

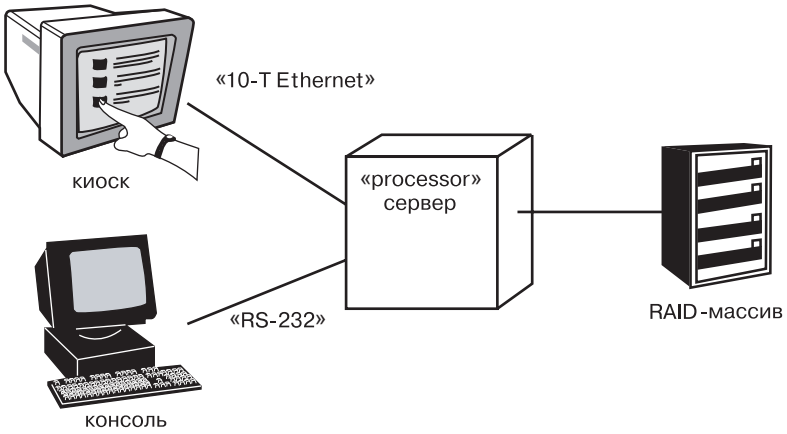


Рис. 27.5. Процессоры и устройства

Моделирование распределения артефактов

Семантика местоположения рассматривается в главе 24.

При моделировании топологии системы бывает полезно визуализировать или специфицировать физическое распределение ее артефактов по процессорам и устройствам, входящим в состав системы.

Моделирование распределения артефактов состоит из следующих шагов:

- Припишите каждый значимый компонент системы к определенному узлу.
- Рассмотрите возможности дублирования размещения артефактов. Довольно распространен случай, когда одни и те же артефакты (например, некоторые исполняемые программы и библиотеки) размещаются одновременно в нескольких узлах.
- Изобразите распределение артефактов по узлам одним из трех способов:

  1. Не делайте размещение видимым, но оставьте его на заднем плане модели, то есть в спецификации каждого узла;
  2. Соедините каждый узел с артефактами, которые на нем размещены, связью зависимости;
  3. Перечислите артефакты, размещенные на узле, в дополнительном разделе.

Экземпляры обсуждаются в главе 13, диаграммы объектов – в главе 14.

Последний из перечисленных способов проиллюстрирован на рис. 27.6, основанном на предыдущих диаграммах. Здесь специфицированы исполняемые артефакты, размещенные в каждом узле. Эта диаграмма несколько отличается от предыдущих – она является диаграммой объектов, на которой визуализированы конкретные экземпляры каждого узла. В данном случае экземпляры `RAID farm`

(RAID-массив) и kiosk (киоск) анонимны, а у остальных двух экземпляров есть имена: c для console (консоль) и s для server (сервер). Для каждого процессора на рисунке отведен дополнительный раздел, показывающий, какие артефакты на нем развернуты. Объект server также изображен со своими атрибутами: processorSpeed (скорость-Процессора) и memory (память), причем их значения видимы. Раздел размещения может содержать список имен артефактов в текстовом виде или вложенные графические символы артефактов.

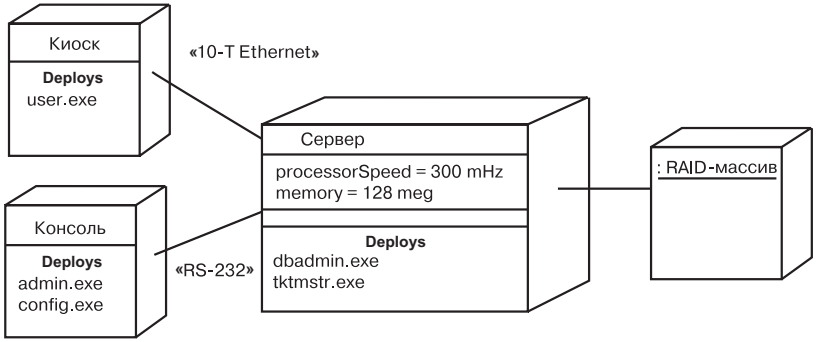


Рис. 27.6. Моделирование распределения артефактов

Советы и подсказки

Хорошо структурированный узел обладает следующими свойствами:

- представляет четкую абстракцию некоторой сущности из словаря аппаратных средств области решения;
- декомпозирован лишь до такого уровня, который необходим, чтобы выразить ваши намерения;
- раскрывает только те атрибуты и операции, которые относятся к моделируемой области;
- явно показывает набор артефактов, которые на нем расположены;
- соединен с другими узлами способом, отражающим топологию реальной системы.

Изображая узел в UML, руководствуйтесь следующими принципами:

- определите для своего проекта или организации в целом набор стереотипов с подходящими пиктограммами, которые несут очевидную для читателя смысловую нагрузку;
- показывайте только те атрибуты и операции (если таковые существуют), которые необходимы для понимания назначения узла в данном контексте.

Глава 28. Кооперации

В этой главе:

В контексте системной архитектуры кооперация позволяет выделить концептуальную часть, которая подчеркивает и статические, и динамические аспекты. Кооперацией называется сообщество классов, интерфейсов и других элементов, которые работают совместно, обеспечивая поведение, представляющее собой нечто большее, чем поведение всех тех же частей в сумме.

Кооперация помогает специфицировать реализацию вариантов использования и операций, а также моделировать архитектурно значимые механизмы системы.

Введение

Представьте самое прекрасное здание, которое вы когда-либо видели, – скажем, Тадж Махал или Собор Парижской Богоматери. Красота обеих построек не поддается описанию. Их архитектура в общем-то проста, хотя заложенные в ней идеи отнюдь не лежат на поверхности. В каждой конструкции легко проглядывается симметрия. Если посмотреть внимательнее, можно заметить подробности, прекрасные сами по себе, которые в сочетании друг с другом создают впечатление гармонии и функциональности – большей, чем таится в каждой отдельной детали.

А теперь вспомните самое нелепое и отвратительное здание из виденных вами – допустим, кафе по соседству с вашим домом. Здесь налицо смешение архитектурных стилей: в общий модернистский дизайн никак не вписывается островерхая крыша, украшения не гармонируют ни с тем, ни с другим, кричащие цвета оскорбляют взгляд. Обычно такие здания – «шедевры» обычных

ремесленников, несущие в себе только идею функциональности, без претензий на стиль.

Что же отличает эти два вида гражданской архитектуры? Во-первых, в величественных зданиях мы видим изящество дизайна, которого явно не хватает забегаловкам. Например, в конструкции Тадж Махала присутствуют сложные, симметричные и сбалансированные геометрические элементы. Вообще, в архитектуре не так уж много стилей, которые удачно сочетаются друг с другом, и талантливый специалист всегда это учитывает. Во-вторых, здания, спроектированные со вкусом, имеют общие структуры, которым подчинены все отдельные элементы конструкции. Так, в Соборе Парижской Богоматери некоторые стены являются несущими и служа т опорой куполу собора. А часть из них наряду с другими архитектурными деталями используется для отвода воды и нечистот.

Пять представлений архитектуры обсуждаются в главе 2. Образцы и каркасы рассматриваются в главе 29.

И опять же, говоря о программном обеспечении, мы сможем провести прямую аналогию с областью строительства. Качественная программная система не только выполняет возложенные на нее функции, но и демонстрирует гармонию, то есть уравновешенность проекта, благодаря чему легко поддается модификации. Эта гармоничность и сбалансированность чаще всего объясняется тем, что хорошо структурированные объектно-ориентированные системы содержат множество повторяющихся структурных *элементов-образцов* (patterns). Если посмотреть на любую качественную объектно-ориентированную систему, то можно обнаружить элементы, взаимодействующие друг с другом для реализации некоторого совместного поведения, представляющего собой нечто большее, чем поведение суммы всех тех же составляющих. Многие элементы хорошо структурированных систем в разных комбинациях принимают участие в функционировании различных механизмов.

Образцы и каркасы обсуждаются в главе 29.

На заметку. Образец описывает общее решение некоторой общей проблемы в определенном контексте. В любой хорошо структурированной системе всегда присутствует целый спектр образцов, включая идиомы (представляющие определенные общие приемы программирования), механизмы (образцы проектирования, представляющие концептуально законченные фрагменты системной архитектуры) и каркасы (архитектурные образцы, представляющие расширяемые шаблоны приложений в определенной области).

В UML механизмы моделируются с помощью кооперации. *Кооперация* именует совокупность взаимодействующих строительных блоков системы, включая как структурные, так и поведенческие элементы. Например, можно рассмотреть распределенную систему

Структурное моделирование обсуждается в частях II и III, моделирование поведения – в частях IV и V. Взаимодействия рассматриваются в главе 16.

управления информацией, база данных которой размещается на нескольких узлах. С точки зрения пользователя обновление информации выглядит как атомарная операция. Если же взглянуть на нее изнутри, все окажется не так просто, поскольку в обновлении данных участвует несколько машин. Для создания иллюзии простоты необходимо ввести механизм транзакций, с помощью которого клиент может присвоить имя некоей операции, которая представляется единой и неделимой, несмотря на то что затрагивает несколько баз данных. В работе такого механизма могли бы принимать участие несколько кооперирующихся классов, совместно обеспечивающих транзакцию. Многие из этих классов будут вовлечены и в другие механизмы – например, в механизм хранения информации. Такой набор классов (структурная составляющая), взятый вместе с взаимодействиями между ними (поведенческая составляющая), образует механизм, который в UML представляется кооперацией.

Варианты использования обсуждаются в главе 17, операции – в главах 4 и 9.

Кооперация не только именует системные механизмы, но и служит в качестве реализации вариантов использования и операций.

Графическое представление кооперации в UML показано на рис. 28.1. Эта нотация позволяет визуализировать структурные и поведенческие строительные блоки системы, особенно в ситуациях, когда они пересекаются с классами, интерфейсами и другими элементами.

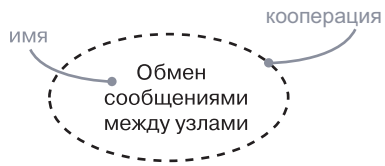


Рис. 28.1. Кооперации

Диаграммы классов обсуждаются в главе 8, диаграммы взаимодействия – в главе 19.

На заметку. Приведенная нотация позволяет визуализировать кооперацию как единый фрагмент с точки зрения внешнего наблюдателя. Но более интересно то, что находится внутри кооперации. Раскройте ее, и вы увидите другие диаграммы, наиболее важные из которых – диаграммы классов (структурная составляющая кооперации) и диаграммы взаимодействия (ее поведенческая составляющая).

Базовые понятия

Кооперация (collaboration) – это сообщество классов, интерфейсов и других элементов, которые работают совместно для обеспечения определенного поведения, более значимого, чем поведение

суммы всех тех же составляющих. Кооперация также показывает, как некий элемент – например, классификатор (включая класс, интерфейс, компонент, узел или вариант использования) либо операция, реализуется набором классификаторов и ассоциаций, каждая из которых определенным образом играет определенную роль. Кооперация изображается в виде эллипса с пунктирной границей.

Нотация кооперации похожа на ту, что применяется для вариантов использования (см. главу 17).

Имена

Имя кооперации должно быть уникальным в пределах включающего пакета (см. главу 12).

Каждая кооперация должна иметь имя, отличающее ее от других кооперация. Имя – это текстовая строка; взятое в отдельности, оно называется *простым*. *Квалифицированное имя* кооперации снабжено префиксом – именем пакета, в котором она находится. Как правило, при изображении кооперации указывается только ее имя, как показано на рис. 28.1.

На заметку. Имя кооперации может состоять из любых букв латинского алфавита, цифр и некоторых знаков препинания (за исключением таких, как двоеточие, которое используется для отделения имени кооперации от имени включающего его пакета). Имя может записываться в несколько строк; количество символов в нем не ограничено. На практике для именования кооперация используют краткие существительные, взятые из словаря моделируемой системы. Обычно первая буква имени кооперации – заглавная, например: `Transaction` (Транзакция), `Chain of responsibility` (Цепочка обязанностей).

Структура

Структурные элементы обсуждаются в частях II и III.

Кооперации имеют две составляющие: структурную, которая описывает классы, интерфейсы и другие совместно работающие элементы, и поведенческую, которая описывает динамику взаимодействия этих элементов.

Структурная составляющая кооперации – это внутренняя (составная) структура, которая может включать любую комбинацию классификаторов, таких как классы, интерфейсы, компоненты, узлы. Внутри кооперации эти классификаторы могут быть организованы с использованием всех обычных связей UML, включая ассоциации, обобщения и зависимости. Фактически структурные аспекты кооперация могут использовать полный диапазон средств структурного моделирования UML.

Однако в отличие от структурированных классов кооперация не владеет своими структурными элементами. Вместо этого

Классификаторы обсуждаются в главе 9, связи – в главах 5 и 10, внутренние структуры – в главе 15, пакеты – в главе 12, подсистемы – в главе 32, варианты использования – в главе 17.

она просто ссылается на классы, интерфейсы, компоненты, узлы и прочие структурные элементы, объявленные в другом месте, или использует их. Вот почему кооперация именует концептуальный, а не физический фрагмент системной архитектуры. Кооперация может распространяться на многие уровни системы. Более того, один и тот же элемент может принимать участие в нескольких кооперациях (а некоторые элементы не будут являться частью ни одной из них).

Например, система розничной торговли через Internet описывается примерно дюжиной вариантов использования. Среди них, в частности, будут такие, как `Purchase Items` (Покупка товара), `Return Items` (Возврат товара), `Query Order` (Просмотр заказа) и т.п., и каждый из них может быть реализован отдельной кооперацией. Вдобавок эти кооперации будут разделять некоторые общие структурные элементы – такие как классы `Customer` (Клиент) и `Order` (Заказ), – но организованные по-разному. На более глубоких уровнях системы также обнаружатся кооперации, представляющие архитектурно значимые механизмы. Скажем, в системе розничной торговли может присутствовать кооперация `Internode messaging` (Межузловые сообщения), которая описывает детали защищенной передачи сообщений между узлами.

Диаграммы классов обсуждаются в главе 8.

Если имеется кооперация, именующая концептуальный фрагмент системы, то вы можете погрузиться в нее, чтобы рассмотреть скрытые внутри структурные детали. Например, на рис. 28.2 показано, какой набор классов, изображенный на диаграмме классов, обнаружится при раскрытии кооперации `Internode messaging`.

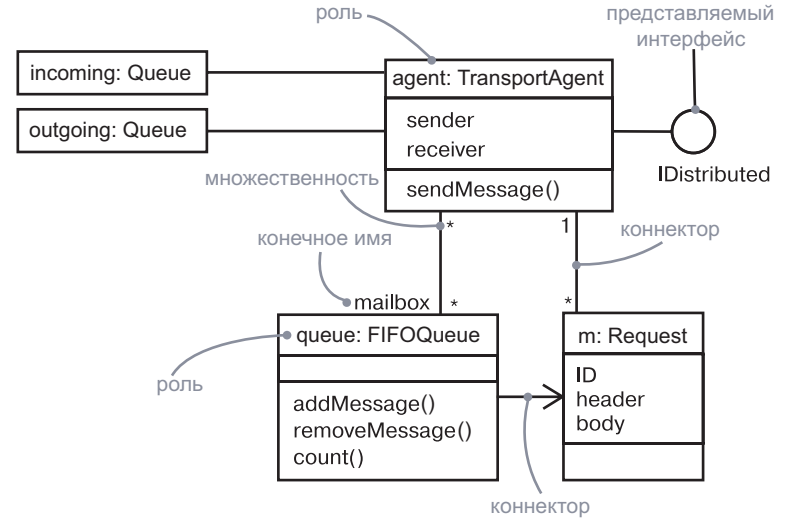


Рис. 28.2. Структурные аспекты кооперации

Поведение

Диаграммы взаимодействия обсуждаются в главе 19, экземпляры – в главе 13, составные структуры – в главе 15.

Если структурная составляющая кооперации обычно изображается как диаграмма составной структуры, то поведенческая в большинстве случаев представлена диаграммой взаимодействия. Эта диаграмма описывает взаимодействие, соответствующее поведению, суть которого – обмен сообщениями между объектами в некотором контексте для достижения определенной цели. Контекст взаимодействия устанавливает сама кооперация, определяющая классы, интерфейсы, компоненты, узлы и другие структурные элементы, экземпляры которых могут принимать участие во взаимодействии.

Поведенческую составляющую кооперации можно описать одной или несколькими диаграммами взаимодействия. Если необходимо подчеркнуть временной порядок сообщений, используйте диаграмму последовательности; если же основной акцент нужно сделать на структурных связях между объектами, возникающих в ходе совместной деятельности, применяйте диаграмму коммуникации. В обоих случаях подходит любой вид диаграмм, поскольку в большинстве случаев они семантически эквивалентны.

Это означает, что, моделируя взаимодействия сообщества классов внутри некоторой кооперации, вы можете раскрыть ее и ознакомиться с подробностями поведения. Так, раскрывая кооперацию `Internode messaging`, вы могли бы увидеть диаграмму взаимодействия, показанную на рис. 28.3.

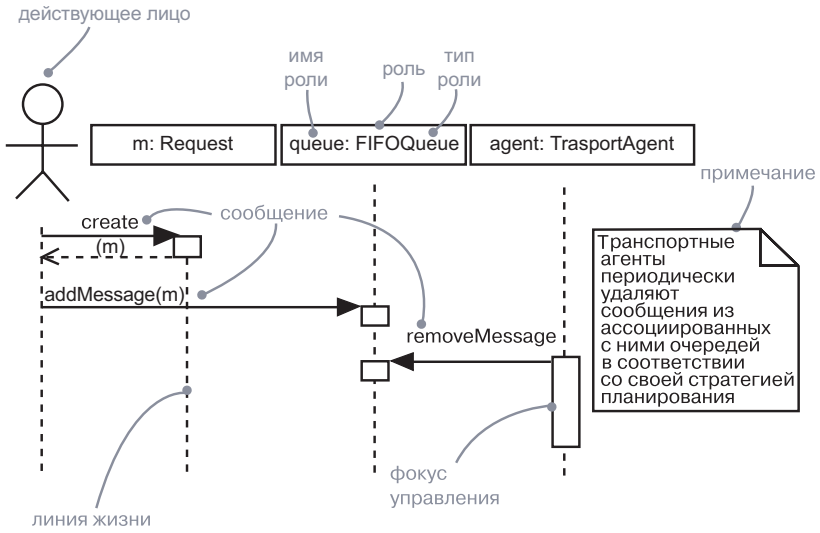


Рис. 28.3. Поведенческие аспекты кооперации

На заметку. Поведенческие части кооперации должны быть согласованы со структурными. Иными словами, объекты, участвующие в кооперативных взаимодействиях, должны быть экземплярами классов, входящих в структурную часть. Аналогичным образом упомянутые во взаимодействии сообщения должны соотноситься с операциями, видимыми в структурной части кооперации. С кооперацией может быть ассоциировано несколько взаимодействий, показывающих разные (но согласованные) аспекты поведения.

Организация коопераций

В кооперациях системы находится сердце ее архитектуры, потому что лежащие в основе системы механизмы представляют существенные проектные решения. Все хорошо структурированные объектно-ориентированные системы состоят из регулярного (то есть построенного в соответствии с четкими правилами) множества коопераций относительно небольшого размера, поэтому очень важно научиться их организовывать. Существует два вида относящихся к кооперациям связей, которые следует принимать во внимание.

Варианты использования рассматриваются в главе 17, операции – в главах 4 и 9, связи реализации – в главе 10, классификаторы – в главе 9.

Во-первых, это связь между кооперацией и тем, что она реализует. Кооперация может реализовывать либо классификатор, либо операцию. Это означает, что кооперация описывает структурную или поведенческую реализацию соответствующего классификатора или операции. Например, вариант использования, который именует набор сценариев, выполняемых системой, может быть реализован в виде кооперации. Этот вариант использования вместе с его действующими лицами и окружающими вариантами использования представляет контекст кооперации. Аналогично кооперацией может быть реализована и операция (которая именует реализацию некоторой системной услуги). В таком случае контекст формирует данная операция вместе со своими параметрами и возможным возвращаемым значением. Такая связь между вариантом использования или операцией и реализующей кооперацией моделируется с помощью связи реализации.

Классификаторы обсуждаются в главе 9.

На заметку. Кооперация может реализовывать любой вид классификатора, включая классы, варианты использования, интерфейсы, компоненты и узлы. Кооперация, которая моделирует механизм системы, может быть и автономной; в таком случае ее контекстом является вся система в целом.

Во-вторых, существуют связи между самими кооперациями. Одни из них могут уточнять описания других; это может быть смоделировано в виде связи уточнения. Подобные связи между кооперациями обычно отражают связи уточнения между вариантами использования, которые они представляют.

Оба вида связей иллюстрирует рис. 28.4.

Рис. 28.4. Организация коопераций

Пакеты обсуждаются в главе 12.

На заметку. Кооперации, как и любые другие элементы моделей UML, могут группироваться в пакеты. Как правило, в этом возникает необходимость только при моделировании очень больших систем.

Типичные приемы моделирования

Моделирование ролей

Объекты представляют отдельные сущности в статическом положении или в динамике. Вместе с тем мы нередко хотим показать некоторые общие части в определенном контексте. Часть в пределах контекста называется *ролью* (role). Вероятно, наиболее важное назначение ролей заключается в моделировании динамических взаимодействий. Обычно в таких случаях не приходится моделировать конкретные экземпляры, существующие в реальном мире.

Взаимодействия обсуждаются в главах 16 и 19. Внутренняя структура рассматривается в главе 15.

Вместо этого вы моделируете роли в некоем повторно используемом образце, в пределах которого они, по сути, замещают объекты, появляющиеся в индивидуальных экземплярах образца. Например, если вы хотите смоделировать варианты реакции окон в операционной системе на движения и щелчки мыши, вы должны нарисовать диаграмму взаимодействий, содержащую роли, типы которых включают окна, события и обработчики.

Чтобы смоделировать роли, необходимо:

- Определить контекст, в котором взаимодействуют объекты.
- Идентифицировать необходимые и достаточные роли, с помощью которых можно визуализировать, специфицировать, конструировать или документировать моделируемый контекст.
- Изобразить их как роли в структурированном контексте. По возможности присвоить каждой роли имя. Если для какой-либо из них нельзя подобрать осмысленное имя, изобразить ее как безымянную.
- Раскрыть свойства каждой роли, которые необходимы и достаточны для моделирования контекста.
- Представить роли и их связи на диаграмме взаимодействия или диаграмме классов.

На заметку. Семантическая разница между конкретными объектами и ролями достаточно тонка, но не сложна для понимания. Уточняя сказанное выше, следует отметить, что роль UML – это предопределенная часть структурированного классификатора, такого как структурированный класс или кооперация. Роль – это не объект, но описание; она привязана к значению внутри каждого экземпляра структурированного классификатора. Таким образом, роль, как и атрибут, соответствует многим возможным значениям. Конкретные объекты появляются в специфических примерах, в частности на диаграммах объектов, компонентов и размещения. Роли появляются в обобщенных описаниях – на диаграммах взаимодействия и деятельности.

Диаграммы взаимодействия обсуждаются в главе 19, диаграммы деятельности – в главе 20.

На рис. 28.5 вы видите диаграмму взаимодействия, описывающую частичный сценарий осуществления телефонного вызова в контексте коммутатора. Здесь налицо четыре роли: `a` (`CallingAgent` – ВызывающийАбонент), `c` (`Connection` – Подключение), а также `t1` и `t2` (экземпляры `Terminal` – Терминал). Все они, по своей сути, являются концептуальными «заместителями» конкретных объектов из реального мира.

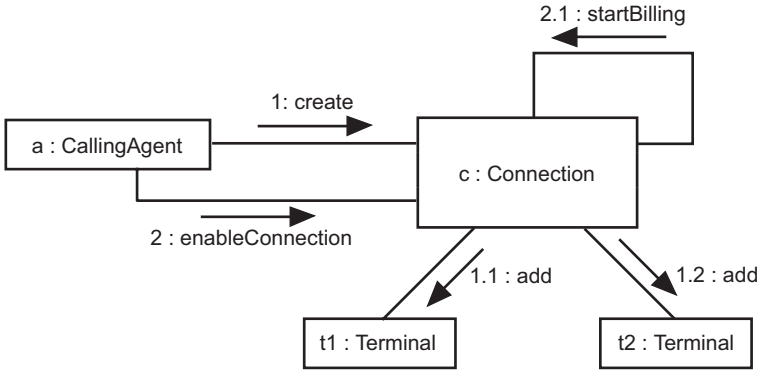


Рис. 28.5. Моделирование ролей

На заметку. На рис. 28.5 представлен пример кооперации, которая представляет сообщество объектов и других элементов, работающих совместно для обеспечения определенного поведения, представляющего собой нечто большее, чем поведение суммы тех же элементов. Кооперации имеют две составляющие: структурную (описывающую роли классификаторов и их связи) и динамическую, или поведенческую (описывающую взаимодействия между этими прототипными экземплярами).

Моделирование реализации варианта использования

Варианты использования обсуждаются в главе 17.

Одно из назначений кооперации состоит в моделировании реализации варианта использования. Как правило, анализ системы диктуется теми вариантами использования, которые вы идентифицировали. Переходя же к этапу реализации, вы должны материализовать их в виде конкретных структур и поведений. В общем случае каждый вариант использования должен быть реализован одной или несколькими кооперациями. Если рассматривать систему в целом, то классификаторы, участвующие в кооперации, которая связана с некоторым вариантом использования, будут принимать участие и в других кооперациях. Таким образом, структурное содержимое кооперации имеет тенденцию перекрываться.

Чтобы смоделировать реализацию варианта использования, следует:

- Идентифицировать структурные элементы, необходимые и достаточные для выражения семантики варианта использования.

- Организовать эти структурные элементы в диаграммы классов.
- Рассмотреть отдельные сценарии, представляющие данный вариант использования. Каждый сценарий описывает конкретный путь выполнения варианта использования.
- Отобразить динамику этих сценариев на диаграммах взаимодействия. Использовать диаграммы последовательности, если нужно подчеркнуть временной порядок сообщений, или диаграммы коммуникации, если важнее структурные связи между объектами, участвующими в кооперации.
- Организовать эти структурные и поведенческие элементы как кооперацию, которую можно соединить с вариантом использования при помощи связи реализации.

Например, на рис. 28.6 изображены варианты использования, относящиеся к системе контроля кредитных карточек, включая основные: `Place order` (Разместить заказ) и `Generate bill` (Выдать счет), а также подчиненные: `Detect card fraud` (Обнаружить мошенничество) и `Validate transaction` (Проверить транзакцию). Хотя в большинстве случаев не возникает необходимости в явном моделировании этой связи – такую задачу можно возложить на инструментальные средства, – на данном рисунке показана явная модель реализации `Place order` с помощью кооперации `Order management` (Управление заказами). В свою очередь, эта кооперация может быть разложена

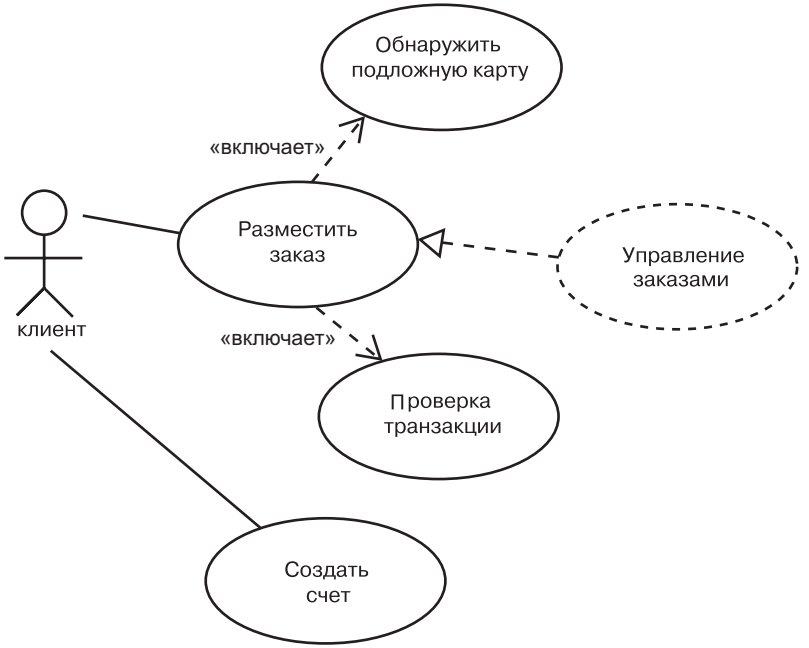


Рис. 28.6. Моделирование реализации варианта использования

на структурный и поведенческий аспекты, что в итоге дает диаграммы классов и диаграммы взаимодействия. Таким образом, через связь реализации вариант использования связывается с его сценариями.

В большинстве случаев нет необходимости моделировать явно связь между вариантом использования и реализующей его кооперацией. Можно оставить это соединение на заднем плане модели, а затем позволить инструментальным средствам использовать его, чтобы упростить навигацию между вариантом использования и его реализацией.

Моделирование реализации операции

Операции обсуждаются в главах 4 и 9.

Еще одно назначение кооперации – это моделирование реализации операций. Часто ее можно специфицировать непосредственно в коде, однако для тех операций, которые требуют совместной работы нескольких объектов, перед написанием кода лучше смоделировать реализацию с помощью кооперации.

Контекст реализации операции составляют параметры, возвращаемые значения и объекты, локальные по отношению к ней. Таким образом, эти элементы видимы для структурного аспекта кооперации, которая реализует данную операцию (подобно тому как действующие лица видимы для структурного аспекта кооперации, реализующей вариант использования). Связи между этими частями можно смоделировать при помощи диаграмм составной структуры, описывающих структурную часть кооперации.

Чтобы смоделировать реализацию операции, необходимо:

Примечания обсуждаются в главе 6.

- Идентифицировать параметры, возвращаемые значения и другие объекты, видимые операции. Они выступят в качестве ролей кооперации.
- Если операция достаточно проста, представить ее реализацию непосредственно в коде, который можно поместить на задний план модели либо явно визуализировать в примечании.
- Если операция сложна алгоритмически, смоделировать ее реализацию с помощью диаграммы деятельности.
- Если операция сложна либо требует долгого и тщательного проектирования, представить ее реализацию в виде кооперации. В дальнейшем можно развернуть структурную и поведенческую составляющие кооперации, используя соответственно диаграммы классов и диаграммы взаимодействия.

В качестве примера на рис. 28.7 показан активный класс `RenderFrame` (ПостроениеФрейма) и раскрыты три его операции. Функция `progress` (обработка) достаточно проста и может быть реализована сразу в коде, приведенном в примечании. А вот операция `render` (визуализировать) намного сложнее, поэтому ее реализация возложена на кооперацию `Ray trace` (Трассировка лучей). Хотя на рисунке это не показано, вы могли бы изучить кооперацию изнутри и увидеть ее структурные и поведенческие аспекты.

Активные классы обсуждаются в главе 23.

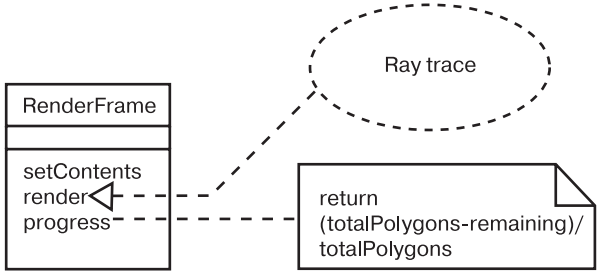


Рис. 28.7. Моделирование реализации операции

Диаграммы деятельности обсуждаются в главе 20.

На заметку. Операцию можно моделировать и с использованием диаграмм деятельности. Эти диаграммы, по сути, представляют собой блок-схемы, поэтому для алгоритмически сложных операций, которые нужно моделировать явно, это обычно наилучший выбор. Однако если операция требует участия множества объектов, лучше выбрать кооперации, поскольку они позволяют моделировать как структурные, так и поведенческие аспекты операции.

Моделирование механизма

Образцы и каркасы обсуждаются в главе 29. Там же рассматривается пример моделирования механизма.

В хорошо структурированной объектно-ориентированной системе всегда присутствует целый спектр образцов. На одном конце этого спектра вы обнаружите идиомы, представляющие устойчивые конструкции языка реализации, а на другом – архитектурные образцы и каркасы, образующие систему в целом и задающие определенный стиль. В середине же спектра располагаются механизмы, описывающие распространенные образцы проектирования, посредством которых системы взаимодействуют друг с другом. Механизмы в UML представляют с помощью кооперации.

Механизмы – это автономные кооперации; их контекстом является не какой-то один вариант использования или одна операция, а система в целом. Любой элемент, видимый в некоторой части системы, – кандидат на участие в механизме.

Механизмы представляют архитектурно значимые проектные решения, и относиться к ним следует серьезно. Обычно механизмы предлагает системный архитектор, и с каждой новой версией они изменяются. В конце концов вы обнаруживаете, что система стала простой (поскольку в механизме реализованы типичные взаимодействия), понятной (так как к пониманию системы можно подойти со стороны ее механизмов) и гибкой (настраивая каждый механизм, вы настраиваете систему в целом).

Чтобы смоделировать механизм, необходимо:

- Идентифицировать основные механизмы, образующие архитектуру системы. Их выбор диктуется общим архитектурным стилем, который вы решили положить в основу своей реализации, наряду со стилем, наиболее подходящим к предметной области.
- Представить каждый механизм в виде кооперации.
- Раскрыть структурную и поведенческую составляющие части каждой кооперации, по возможности стараясь находить совместно используемые элементы.
- Утвердить механизмы на ранней стадии жизненного цикла разработки (они имеют стратегическое значение), но впоследствии в каждой новой версии развивать их по мере прояснения деталей реализации.

Советы и подсказки

При моделировании кооперации в UML помните, что каждая из них должна либо представлять реализацию варианта использования, либо служить автономным механизмом уровня всей системы. Хорошо структурированная кооперация обладает следующими свойствами:

- включает как структурный, так и поведенческий аспекты;
- представляет собой четкую абстракцию некоторого идентифицируемого взаимодействия в системе;
- редко бывает полностью независимой, но чаще перекрывается со структурными элементами других кооперации;
- понятна и проста.

Изображая кооперацию в UML, придерживайтесь следующих правил:

- явно изображайте кооперацию только тогда, когда это необходимо для понимания ее связей с другими кооперациями, классификаторами, операциями или системой в целом. В остальных случаях используйте кооперации, но оставляйте их на заднем плане модели;
- организуйте кооперации в соответствии с представленными ими классификаторами или операциями либо помещайте в пакеты, ассоциированные с системой в целом.

Глава 29. Образцы и каркасы

В этой главе:

Все хорошо структурированные системы полны образцов (patterns). Образец предлагает типичное решение типичной проблемы в данном контексте. Механизм – это образец проектирования, применяемый к сообществу классов; каркас (framework) – это, как правило, архитектурный образец, предлагающий расширяемый шаблон для приложений в некоторой предметной области.

Образцы используются для специфицирования механизмов и каркасов, образующих архитектуру системы. Вы делаете образец доступным, ясно идентифицируя все «вилки», «розетки», «кнопки» и «циферблаты», с помощью которых пользователь настраивает образец для применения его в определенном контексте.

Введение

О том, сколько существует способов собрать дом из кучи досок, можно было бы исписать тысячи страниц. Мастер из Сан-Франциско представит на ваш суд выкрашенное в яркий цвет здание в викторианском стиле, с остроконечной крышей. Строитель из штата Мэн превратит груду досок в домик в виде солонки, обшитой дранкой.

Внешне эти дома представляют собой два совершенно разных архитектурных стиля. Каждый строитель, основываясь на собственном опыте, выберет тот стиль, который лучше всего удовлетворяет требованиям заказчика, а затем слегка подкорректирует с учетом пожеланий клиента и ограничений, накладываемых выбранным местом, а также с местными строительными нормами и правилами.

При проектировании дома нельзя снять со счетов и некоторые общие проблемы, устанавливающие разумные пределы архитекторским фантазиям. Есть лишь ограниченное число проверенных способов конструирования стропил, на которые опирается крыша, и ограниченное число способов проектирования несущей стены с дверными и оконными проемами. Каждому строителю для решения этих типичных проблем придется выбрать те или иные механизмы, приспосабливая их к общему архитектурному стилю и местным нормам.

Построение программных систем подчиняется тем же принципам. Всякий раз, отрывая глаза от конкретных строк кода, вы обнаруживаете типичные механизмы, которые определяют способ организации классов и других абстракций. Например, в управляемой событиями системе применение образца проектирования, известного под именем «цепочка обязанностей», – это типичный способ организации обработчика событий. Если взглянуть чуть выше уровня этих механизмов, то вы увидите типичные каркасы, формирующие архитектуру всей системы. Например, в информационных системах применение трехслойной архитектуры – распространенный способ достижения четкого разделения обязанностей между пользовательским интерфейсом, хранением информации и бизнес-объектами и правилами.

Кооперации обсуждаются в главе 28, пакеты – в главе 12.

В UML вам нередко придется моделировать образцы проектирования, также называемые механизмами, которые можно представить в виде кооперации. Аналогичным образом архитектурные образцы моделируются как каркасы, представляемые в виде пакетов со стереотипами.

Для образцов обоих типов в UML предусмотрено особое графическое представление (см. рис. 29.1).

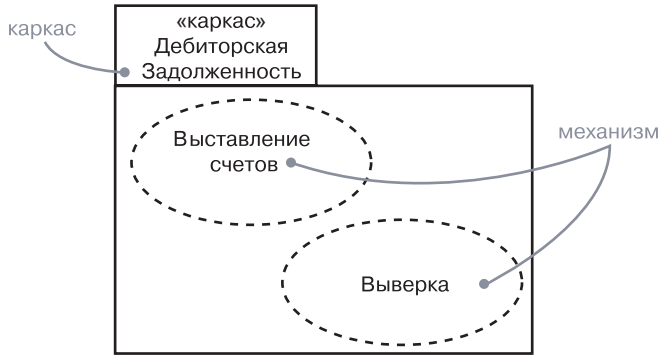


Рис. 29.1. Механизмы и каркасы

Базовые понятия

Образец (pattern) – это общее решение типичной проблемы в данном контексте. *Механизм* – это образец проектирования, применимый к сообществу классов. *Каркас* (framework) – это архитектурный образец, предлагающий расширяемый шаблон для приложений в некоторой предметной области.

Образцы и архитектура

Программная архитектура обсуждается в главе 2.

Занимаясь разработкой архитектуры новой системы или развитием существующей, вы в любом случае никогда не начинаете с нуля. Напротив, прежний опыт и соглашения наталкивают вас на применение типичных приемов решения типичных проблем. Например, при построении системы, активно взаимодействующей с пользователем, вы можете задействовать испытанный образец «модель–представление–контроллер» (model–view–controller), который позволяет четко отделить объекты (модель) от их внешнего представления и от агента, обеспечивающего их синхронизацию (контроллер). А при создании систем дешифровки проверенным способом организации системы окажется применение архитектуры «классной доски» (blackboard), хорошо приспособленной для решения сложных задач методом проб и ошибок.

То и другое представляет пример образца – типичного решения типичных задач в данном контексте. Любая хорошо структурированная система включает в себя множество образцов на различных уровнях абстракции. Образцы проектирования описывают структуру и поведение сообщества классов, а архитектурные образцы – структуру и поведение системы в целом.

Образцы входят в UML просто потому, что являются важной составляющей словаря разработчика. Явно выделяя их в системе, вы делаете ее более понятной и простой в сопровождении. Например, если вам дадут неизвестный исходный текст и попросят его модифицировать, вы потратите много времени, пытаясь догадаться, как его части связаны друг с другом. Вместе с тем, если вам дадут тот же текст и скажут, что перечисленные классы взаимодействуют на основе механизма «публикации-и-подписки» (publish-and-subscribe), вы получите гораздо более ясное представление о том, как все работает. Та же идея применима к системе в целом. Одна лишь фраза «система организована как набор конвейеров и фильтров» очень многое говорит о системной архитектуре – понять это, глядя на код классов, было бы куда сложнее.

Образцы помогают визуализировать, специфицировать, конструировать и документировать артефакты программной системы. Можно заниматься прямым проектированием системы, выбирая подходящий набор образцов и применяя их к абстракциям, специфичным для данной предметной области, или же обратным проектированием, выявляя содержащиеся в системе образцы (хотя вряд ли этот процесс можно назвать очень продуктивным). Впрочем, при поставке системы было бы еще лучше описать характерные для нее образцы, чтобы помочь тому, кому придется в будущем повторно использовать или модифицировать ваш код.

На практике интерес представляют только два вида образцов – образцы проектирования и каркасы. В UML предусмотрены средства моделирования и тех и других. При моделировании любого образца вы обнаружите, что он, как правило, является автономным в некотором большом пакете, если не считать зависимостей, связывающих этот образец с остальными частями системы.

Механизмы

Механизм – это образец проектирования, примененный к сообществу классов. Например, существует типичная проблема проектирования, с которой сталкивается программист, пишущий на языке Java: как изменить класс, который умеет реагировать на некоторое множество событий, таким образом, чтобы он реагировал на события иного рода, не затрагивая исходного кода этого класса? Типичное решение проблемы – применение образца адаптера (adaptor pattern), структурного образца проектирования, который конвертирует один интерфейс в другой. Этот образец является настолько общим, что имеет смысл дать ему название, а затем использовать в моделях всякий раз, когда возникает аналогичная проблема.

При моделировании механизмы проявляют себя двойственно.

Кооперации обсуждаются в главе 28

Во-первых, как было показано на рис. 29.1, механизм просто именует набор абстракций, работающих вместе для реализации типичного поведения, представляющего некоторый интерес. Такие механизмы моделируются как простые кооперации, поскольку они являются всего лишь именами для сообщества классов. Раскрыв такую кооперацию, можно увидеть ее структурные аспекты (обычно изображаемые на диаграмме классов), а также поведенческие аспекты (обычно изображаемые на диаграммах взаимодействия). Кооперации подобного типа охватывают разные уровни абстракции системы, то есть какой-то конкретный класс, вероятно, будет участвовать в нескольких кооперациях.

Во-вторых, как показано на рис. 29.2, механизм именует шаблон для набора абстракций, работающих совместно для обеспечения

Шаблонные классы рассматриваются в главе 9.

некоторого типичного поведения, представляющего интерес. Такие механизмы моделируются в виде параметризованных кооперации, которые изображаются в UML подобно шаблонным классам. Если раскрыть такую кооперацию, можно увидеть ее структурные и поведенческие аспекты. Если свернуть ее, то можно увидеть, как образец применяется к системе, связывая шаблонные части кооперации с существующими абстракциями системы. При моделировании механизма в виде параметризированной кооперации вы описываете все эти «вилки», «розетки», «кнопки» и «циферблаты», с помощью которых можно адаптировать образец, меняя значение его параметров. Подобные кооперации могут появляться в разных частях системы и связываться с различными абстракциями. В приведенном примере классы образца `Subject` (Субъект) и `Observer` (Наблюдатель) связаны с конкретными классами `CallQueue` (ОчередьЗадач) и `SlideBar` (Ползунок) соответственно.

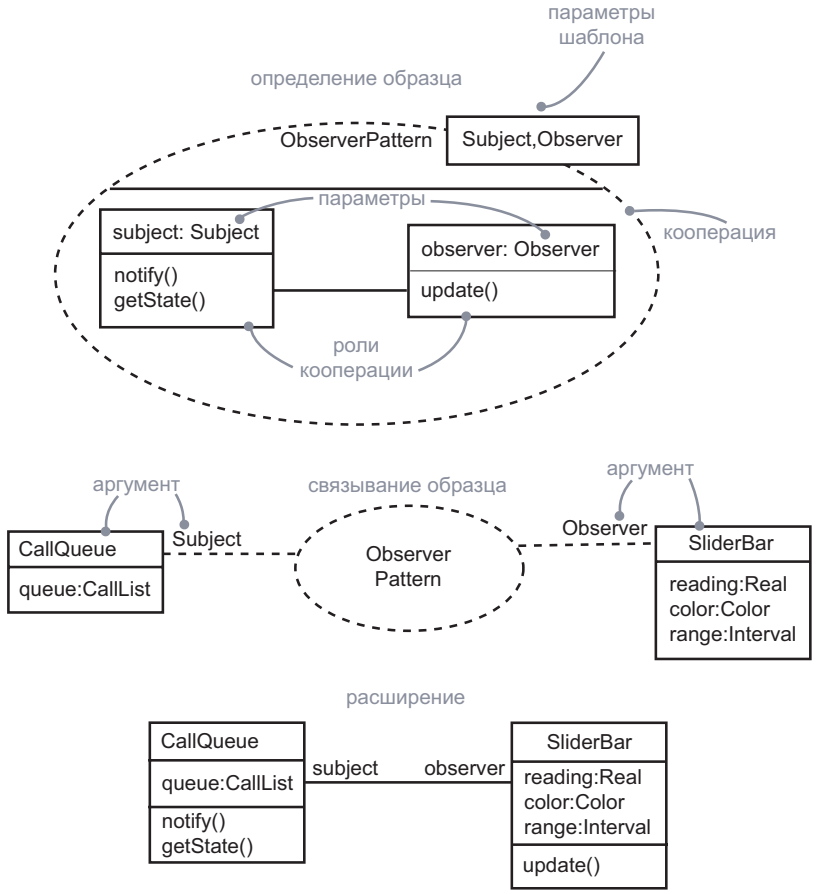


Рис. 29.2. Механизмы

На заметку. Решение о моделировании механизма в виде простой или параметризованной кооперации достаточно очевидно. Если нужно просто именовать сообщество совместно работающих классов в системе, следует применить простую кооперацию. Если же необходимо абстрагировать существенные структурные и поведенческие аспекты механизма способом, не зависящим от конкретной предметной области, а затем связать их с абстракциями в данном контексте, следует применять параметризованную кооперацию.

Каркасы

Каркас (framework) – это архитектурный образец, предлагающий расширяемый шаблон для приложений в некоторой предметной области. Например, в системах реального времени часто можно встретить архитектурный образец «циклический исполнитель» (cyclic executive), который разделяет время на кадры и подкадры, где обработка происходит в строгих временных рамках. Выбор этого образца вместо управляемой событиями архитектуры оказывает влияние на всю систему. Данный образец (как и его альтернатива) является настолько общим, что имеет смысл назвать его каркасом.

Пять представлений архитектуры обсуждаются в главе 2.

Каркас – это нечто большее, чем механизм. Фактически можно считать каркас разновидностью микроархитектуры, включающей в себя множество механизмов, совместно работающих над решением типичной проблемы для типичной предметной области. Специфицируя каркас, вы тем самым описываете «скелет» архитектуры вместе со всеми ее органами управления, которые применяются пользователем для адаптации к нужному контексту.

Пакеты рассматриваются в главе 12, стереотипы – в главе 6.

В UML каркас моделируется в виде пакета со стереотипом. Заглянув внутрь этого пакета, можно увидеть механизмы, существующие в любом из представлений системной архитектуры. Например, там обнаружатся не только параметризованные кооперации, но также варианты использования (которые объясняют, как надо работать с этим каркасом), а также простые кооперации (представляющие набор абстракций, на базе которых можно строить систему, – например, путем порождения классов-потомков).

События обсуждаются в главе 21.

Рис. 29.3 показывает такой каркас, названный `CyclicExecutive` (ЦиклическийИсполнитель). Помимо прочего, этот каркас включает кооперацию `CommonEvents` (ОбщиеСобытия), охватывающую множество классов событий, и механизм `EventHandler` (ОбработчикСобытий), предназначенный для циклической обработки событий. Клиент, построенный на базе этого каркаса, – к примеру, `Pacemaker` (СердечныйСтимулятор), – может пользоваться абстракциями из кооперации `CommonEvents` путем порождения производных классов, а также применять механизм `EventHandler`.

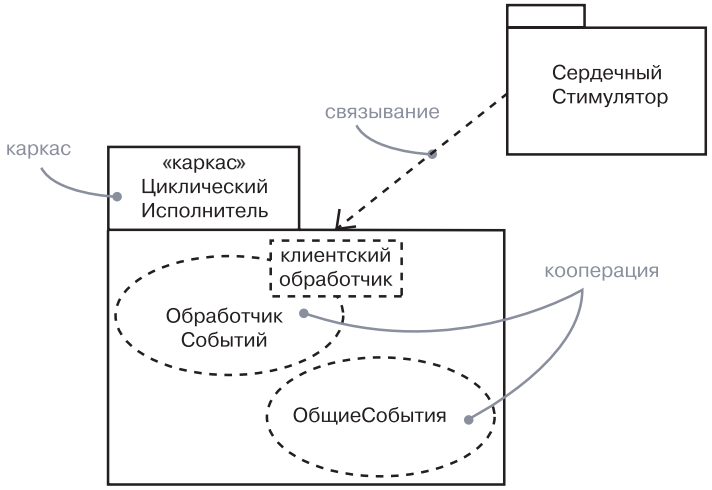


Рис. 29.3. Каркасы

На заметку. Каркасы отличаются от обычных библиотек классов. Библиотека классов содержит абстракции, конкретизируемые или вызываемые другими абстракциями программы. Каркас же содержит абстракции, которые сами вызывают или конкретизируют другие абстракции. Оба вида соединений образуют те самые «вилки», «розетки», «кнопки» и «циферблаты», посредством которых каркас настраивается на используемый вами контекст.

Типичные приемы моделирования

Моделирование образцов проектирования

В числе прочего образцы используются для моделирования типичных ситуаций, возникающих при проектировании. При моделировании подобного механизма следует принимать во внимание его внутренний и внешний вид.

При взгляде снаружи образец проектирования изображается в виде параметризованной кооперации. Будучи кооперацией, образец представляет собой набор абстракций, структура и поведение которых призваны в ходе совместной работы выполнить некоторую полезную функцию. Параметры кооперации именуют те элементы, которые пользователь образца должен с чем-то связать. Таким образом, образец проектирования превращается в шаблон, который используется в конкретном контексте путем подстановки элементов, соответствующих параметрам шаблона.

При взгляде изнутри образец проектирования представляется простой кооперацией и отображается со своими структурной и поведенческой составляющими. Обычно кооперация моделируется с помощью диаграмм классов (для структурной составляющей) и диаграмм взаимодействий (для поведенческой составляющей). Параметры кооперации именуют некоторые из структурных элементов, которые при связывании с каким-то определенным контекстом конкретизируются абстракциями из этого контекста.

Применение кооперации для моделирования механизмов обсуждается в главе 28.

Чтобы смоделировать образец проектирования, необходимо:

- Идентифицировать типичное решение типичной проблемы и материализовать его в виде механизма.
- Смоделировать механизм как кооперацию, описав ее структурный и поведенческий аспекты.
- Идентифицировать те элементы образца проектирования, которые должны быть связаны с элементами в конкретном контексте, и отобразить их в виде параметров кооперации.

В качестве примера на рис. 29.4 показано использование образца проектирования `Command` (см. Gamma et. al. Design Patterns. – Reading, Massachusetts: Addison-Wesley, 1995)[1]. Как установлено в описании данного образца, он «инкапсулирует запрос в виде объекта, позволяя тем самым параметризовать клиентов, выдающих разные запросы, ставить запросы в очередь и протоколировать их, а также поддерживать операции, допускающие отмену». Как видно из модели, этот образец проектирования имеет три параметра, которые, будучи применены к нему, должны быть связаны с элементами в данном контексте. Модель демонстрирует два соединения, в которых классы `PasteCommand` (Команда Вставки) и `OpenCommand` (Команда Открытия) привязываются к параметрам образца.

Здесь параметрами являются `AbstractCommand` (АбстрактнаяКоманда), которая должна быть связана с таким же абстрактным суперклассом в каждом случае, `ConcreteCommand` (КонкретнаяКоманда), которая связывается с различными конкретными классами в разных случаях связывания, и `Receiver` (Приемник), связываемый с классом, над которым команда осуществляет действие. Класс `Command` (Команда) может быть создан образцом, но оформление его в виде параметра позволяет создавать множество иерархий команд.

Заметьте, что `PasteCommand` и `OpenCommand` являются подклассами `Command`. Весьма вероятно, что ваша система будет использовать этот образец много раз с разнообразными связываниями. Возможность такого повторного использования образца проектирования делает процесс разработки на основе образцов весьма эффективным.

[1] Русскоязычное издание книги: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001.

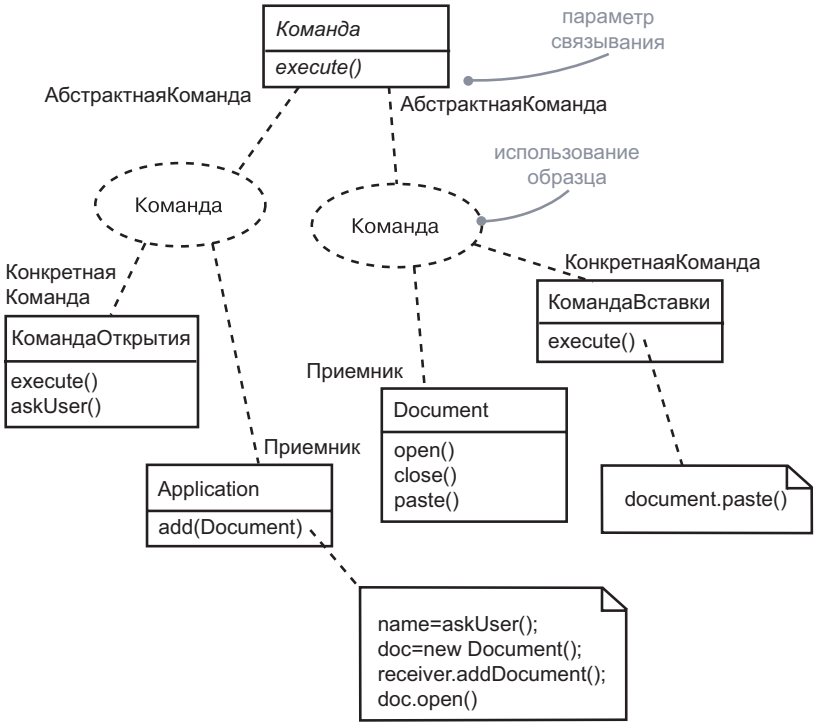


Рис. 29.4. Моделирование образца проектирования

Чтобы завершить модель образца проектирования, вы должны специфицировать его структурную и поведенческую составляющие, которые представляют собой внутренности кооперации.

Кооперации обсуждаются в главе 28, диаграммы классов – в главе 8, диаграммы взаимодействия – в главе 19.

Рассмотрим рис. 29.5, где показана диаграмма классов, представляющая структуру этого образца. Обратите внимание на то, как

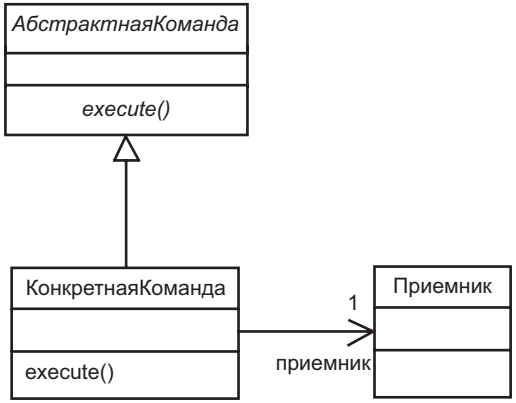


Рис. 29.5. Моделирование структурного аспекта образца проектирования

на этой диаграмме используются классы, которые названы параметрами образца. Рис. 29.6 показывает диаграмму последовательности, демонстрирующую поведение того же образца. Отметим, что она носит характер предположения: образец проектирования не является жесткой конструкцией.

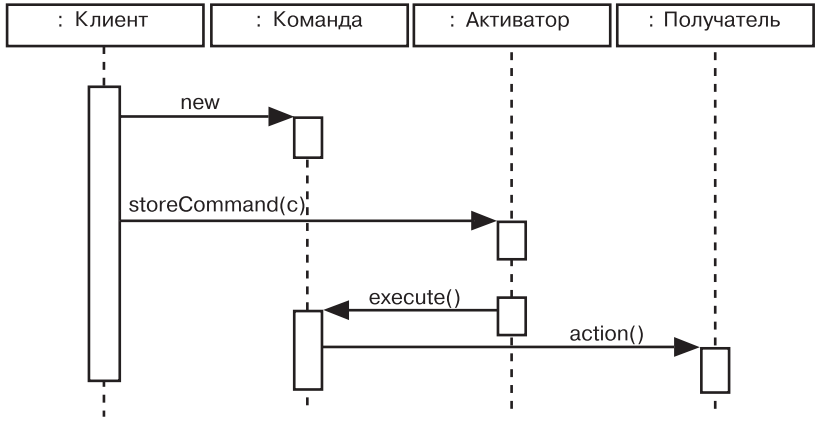


Рис. 29.6. Моделирование поведенческого аспекта образца проектирования

Моделирование архитектурных образцов

Пакеты обсуждаются в главе 12.

Образцы также подходят для моделирования типичных архитектурных решений. При моделировании каркаса вы, собственно, разрабатываете инфраструктуру всей архитектуры, которую затем планируете повторно использовать и адаптировать к некоему контексту.

Каркас изображается в виде пакета со стереотипом. Являясь пакетом, каркас представляет ряд элементов, включая классы, интерфейсы, варианты использования, компоненты, узлы, кооперации и даже другие каркасы (но не ограничиваясь ими). Фактически вы помещаете в каркас все абстракции, которые, работая совместно, формируют расширяемый шаблон для приложений в определенной области. Некоторые из этих элементов будут открыты и станут ресурсами, доступными для использования клиентами. Это те части каркаса, которые вы можете подключать к абстракциям своего контекста. Некоторые из таких открытых элементов станут образцами проектирования и будут представлять собой ресурсы, с которыми связываются клиенты. Именно эти части каркаса вы наполняете, связывая с образцом проектирования. И наконец, некоторые

элементы будут закрытыми или защищенными; они соответствуют инкапсулированным элементам каркаса, не видимым снаружи.

Программная архитектура обсуждается в главе 2.

При моделировании архитектурного образца следует помнить о том, что образец, по сути, является описанием архитектуры, хотя и неполным, и, возможно, параметризованным. Следовательно, все, что вы знаете о моделировании хорошо структурированной архитектуры, в полной мере применимо и к хорошо структурированным каркасам. Они не проектируются в отрыве от остальной системы, – такая попытка обречена на неудачу. В основе каркасов лежат уже существующие архитектуры, доказавшие свою работоспособность. Затем каркасы развиваются, чтобы найти те элементы управления и стыковки, которые необходимы и достаточны для того, чтобы обеспечить возможность их адаптации к новым областям.

Чтобы смоделировать архитектурный образец, необходимо:

- Заложить в основу каркаса проверенную существующую архитектуру.
- Смоделировать каркас как пакет со стереотипом, содержащий все элементы (и особенно образцы проектирования), которые необходимы и достаточны для описания разных представлений каркаса.
- Раскрыть включаемые элементы, интерфейсы и параметры, необходимые для адаптации каркаса, в форме образцов проектирования и кооперации. Это делается с той целью, чтобы пользователь образца понимал, какие классы должны быть расширены, какие операции реализованы и какие сигналы обработаны.

На рис. 29.7 показана спецификация архитектурного образца `Blackboard` (Классная Доска), который позаимствован из книги Buschmann et al. Pattern-Oriented Software Architecture. – New York, NY: Wiley, 1996. Как говорится в его описании, этот образец «применим к задачам преобразования данных в высокоуровневые структуры, которые не имеют простого детерминированного решения». В основе архитектуры лежит образец `Blackboard`, определяющий порядок совместной работы классов `KnowledgeSource` (Источник Знаний), `Blackboard` и `Controller` (Контроллер). Этот каркас включает также образец проектирования `Reasoning engine` (Процессор логического ввода), который определяет общий механизм работы класса `KnowledgeSource`. И наконец, как видно из рисунка, каркас раскрывает один вариант использования – `Apply new knowledge sources` (Применить новые источники знаний), поясняющий клиенту, как можно этот каркас адаптировать.

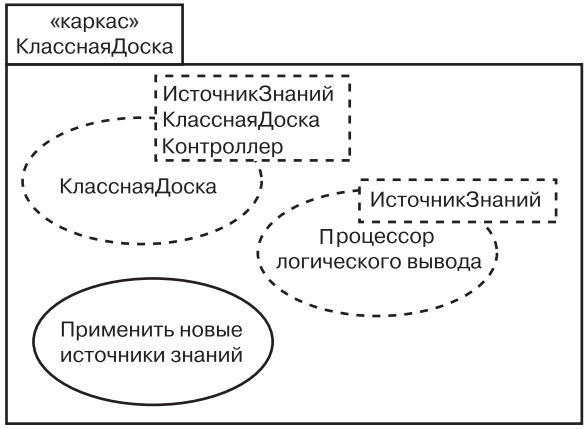


Рис. 29.7. Моделирование архитектурного образца

На заметку. На практике полное моделирование каркаса – задача, по своей сложности сопоставимая с моделированием архитектуры всей системы. В некоторых отношениях она даже сложнее, поскольку для того, чтобы с каркасом можно было работать, вы должны раскрыть все его элементы управления и стыковки и, возможно, представить метаварианты использования (типа `Apply new knowledge sources`), которые показывают, как настраивается каркас, а также простые варианты использования, поясняющие его поведение.

Советы и подсказки

При моделировании образцов в UML следует помнить, что они работают на многих уровнях абстракции, начиная от отдельных классов и заканчивая системой в целом. Самые интересные виды образцов – это механизмы и каркасы. Хорошо структурированный образец обладает следующими свойствами:

- решает типичную проблему типичным образом;
- включает структурную и поведенческую составляющие;
- раскрывает элементы управления и стыковки, с помощью которых его можно настроить на разные контексты;
- охватывает различные индивидуальные абстракции в системе.

Изображая образец в UML, необходимо:

- раскрывать те его элементы, которые следует адаптировать для применения в конкретном контексте;
- приводить варианты использования образца, а также способы его адаптации.

Глава 30. Диаграммы артефактов

В этой главе:

Диаграммы размещения – второй вид диаграмм, используемых для моделирования физических аспектов объектно-ориентированных систем, – обсуждаются в главе 31.

Диаграммы артефактов – это один из двух видов диаграмм, предназначенных для моделирования физических аспектов объектно-ориентированных систем. Диаграмма артефактов показывает организацию и зависимости между наборами артефактов.

Такие диаграммы используются для моделирования статического представления реализации системы, включая моделирование физических сущностей, размещаемых на узлах (например, исполнимых программ, библиотек, таблиц, файлов, документов разного рода). Диаграммы артефактов – это, по сути, диаграммы классов, которые сосредоточены на артефактах системы.

Диаграммы артефактов важны не только для визуализации, специфицирования и документирования систем, основанных на артефактах, но также для конструирования исполняемых систем посредством прямого и обратного проектирования.

Введение

Строительство дома не ограничивается созданием комплекта чертежей. Они, конечно, очень важны, так как помогают визуализировать, специфицировать и документировать, какой именно дом вы собираетесь построить, и обеспечить выполнение замысла с соблюдением сроков и сметы. Но рано или поздно поэтажные планы и разрезы придется воплощать в реальном каркасе из реальных материалов – дерева, камня, металла. При этом вы, скорее всего, будете использовать и уже готовые артефакты, например встроенные шкафы, окна, двери и вентиляционные решетки. А если вы

переоборудуете здание, то число готовых артефактов возрастет, – это будут целые комнаты и инженерные конструкции.

То же самое касается программного обеспечения. Для того чтобы можно было рассуждать о желаемом поведении системы, создаются диаграммы вариантов использования. Словарь предметной области описывается диаграммами классов. Чтобы стало ясно, как сущности из этого словаря совместно работают для обеспечения нужного поведения, применяются диаграммы последовательности, коммуникации, состояний и деятельности. В конечном счете логические чертежи превращаются в реалии из мира битов – исполняемые программы, библиотеки, таблицы, файлы и различные документы. При этом обнаруживается, что некоторые из этих артефактов приходится создавать «с нуля», но находятся и способы повторного использования старых артефактов.

В UML диаграммы артефактов используются в целях визуализации статического аспекта физических артефактов и их связей, а кроме того, описания их деталей для конструирования, как показано на рис. 30.1.

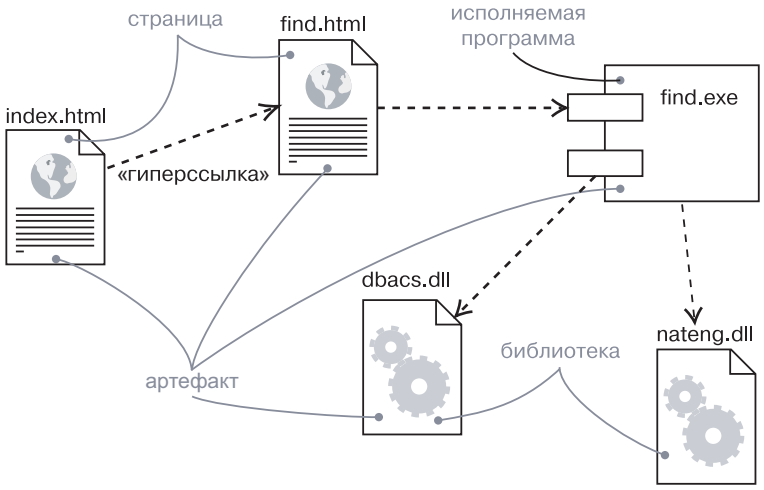


Рис. 30.1. Диаграмма артефактов

Термины и понятия

Диаграмма артефактов показывает набор артефактов и связей между ними. Изображается в виде графа с вершинами и дугами (ребрами).

Общие свойства

Общие свойства всех диаграмм обсуждаются в главе 7.

У диаграммы артефактов, как и у всех диаграмм, есть имя и графическое наполнение. От остальных типов диаграмм она отличается своим содержимым.

Содержимое

Артефакты обсуждаются в главе 26, интерфейсы – в главе 11, связи – в главах 5 и 10, пакеты – в главе 12, подсистемы – в главе 32, экземпляры – в главе 13, диаграммы классов – в главе 8, представления реализации в контексте системной архитектуры – в главе 2.

Диаграммы артефактов обычно содержат артефакты, а также связи зависимости, обобщения, ассоциации и реализации. Подобно другим диаграммам, могут содержать примечания и ограничения.

Общее применение

Диаграммы артефактов применяются для моделирования статического представления реализации системы. Это представление в первую очередь поддерживает управление конфигурацией частей системы, состоящих из артефактов, которые могут быть собраны разными способами для построения работающей системы.

Когда моделируется статическое представление реализации системы, то диаграммы артефактов обычно используются одним из следующих четырех способов:

1. *Для моделирования исходного кода.* В большинстве современных объектно-ориентированных языков программирования код пишется в интегрированных средах разработки, сохраняющих исходные тексты в файлах. Диаграммы артефактов можно применять для моделирования конфигурации этих файлов, которые представляют собой рабочие продукты, и установки вашей системы управления конфигурацией.
2. *Для моделирования исполняемых версий. Версия* (release) – это относительно полный и согласованный набор артефактов, поставляемый внутреннему или внешнему пользователю. Версия в данном понимании сосредоточена на тех частях, которые необходимы для поставки работающей системы. При моделировании версий с помощью диаграмм артефактов происходят визуализация, специфицирование и документирование решений, принятых относительно физических составляющих системы, то есть артефактов размещения.

Свойство сохраняемости, или устойчивости (persistence), обсуждается в главе 24, моделирование логических схем баз данных – в главе 8.

3. *Для моделирования физических баз данных.* Представьте себе физическую базу данных как конкретную реализацию схемы, существующую в мире битов. Схемы, по сути, описывают API для доступа к хранимой информации; модель же физической базы представляет способы хранения информации в таблицах реляционной базы или на страницах объектно-ориентированной базы данных. Для представления этих и иных физических баз данных можно использовать диаграммы артефактов.
4. *Для моделирования адаптируемых систем.* Некоторые системы достаточно статичны: их компоненты появляются на сцене, принимают участие в исполнении, а затем покидают ее. Другие системы более динамичны: они включают в себя мобильных агентов, или артефакты, мигрирующие с целью балансирования нагрузки и восстановления после сбоев. Поведение таких систем моделируется диаграммами артефактов совместно с некоторыми другими диаграммами UML.

Типичные приемы моделирования

Моделирование исходного кода

При разработке программ на языке Java исходный код обычно сохраняется в файлах с расширением .java. Программы, написанные на C++, обычно хранят исходный код в заголовочных файлах с расширением .h и файлах реализации с расширением .cpp. При использовании языка IDL для разработки приложений COM+ или CORBA, единственный, с точки зрения проектирования, интерфейс распадается на четыре исходных файла: сам интерфейс, клиентский заместитель (proxy), серверную заглушку (stub) и класс-мост (bridge). По мере роста объема приложения, на каком бы языке оно ни было написано, эти файлы приходится организовывать в группы. Затем, на стадии сборки приложения, вы, вероятно, станете создавать различные варианты одних и тех же файлов для каждой новой промежуточной версии и захотите поместить все это в систему управления конфигурацией.

Стереотип `file` для артефактов обсуждается в главе 26.

В большинстве случаев нет необходимости моделировать данный аспект системы непосредственно. Вместо этого вы позволите среде разработки отслеживать эти файлы и их связи. Иногда, однако, небесполезно визуализировать исходные файлы и связи между ними с помощью диаграмм артефактов. Применяемые таким образом диаграммы артефактов обычно содержат только артефакты – рабочие продукты со стереотипами файлов – вместе

с их зависимостями. Например, вы могли бы выполнить обратное проектирование набора исходных файлов для визуализации сложной системы зависимостей между ними при компиляции. Можно пойти и в другом направлении, специфицировав связи между исходными файлами и затем передав эту модель на вход инструменту компиляции, такому как программа make в UNIX. Аналогично можно использовать диаграммы артефактов для визуализации истории набора исходных файлов, хранящихся в системе управления конфигурацией. Получая из этой системы информацию (например, о том, сколько раз некий файл извлекался для редактирования на протяжении определенного периода времени), вы можете использовать ее для «раскраски» диаграммы артефактов, что поможет выявить среди исходных файлов «горячие точки», в которых архитектура системы чаще всего подвергается модификациям.

Чтобы смоделировать исходный код системы, необходимо:

- Применяя прямое или обратное проектирование, идентифицировать набор представляющих интерес файлов исходного кода и смоделировать их как артефакты со стереотипом `file`.
- Для крупных систем использовать пакеты, чтобы показать группы исходных файлов.
- Рассмотреть возможность показа с помощью помеченных значений такой информации, как номер версии файла, его автор, дата последнего изменения. Для управления этими значениями применять инструментальные средства.
- Смоделировать зависимости компиляции между исходными файлами с помощью связей зависимости. Опять же, для того чтобы сгенерировать эти зависимости и управлять ими, понадобятся инструментальные средства.

Стереотип зависимости `trace` обсуждается в главе 10.

В качестве примера на рис. 30.2 показаны пять файлов исходного кода. Файл `signal.h` – заголовочный. Представлены три его версии, начиная с последней и заканчивая первой. Каждая версия помечена значением, указывающим ее номер.

Заголовочный файл `signal.h` используется двумя другими – `interp.cpp` и `signal.cpp`. Первый из них при компиляции зависит от другого заголовочного файла, `irq.h`. В свою очередь, файл `device.cpp` зависит от `interp.cpp`. Имея перед глазами такую диаграмму, легко проследить, что произойдет в случае изменений. В частности, изменение исходного файла `signal.h` потребует перекомпиляции трех других файлов – `signal.cpp`, `interp.cpp` и, как следствие, `device.cpp`. Из той же диаграммы видно, что `irq.h` не коснутся преобразования.

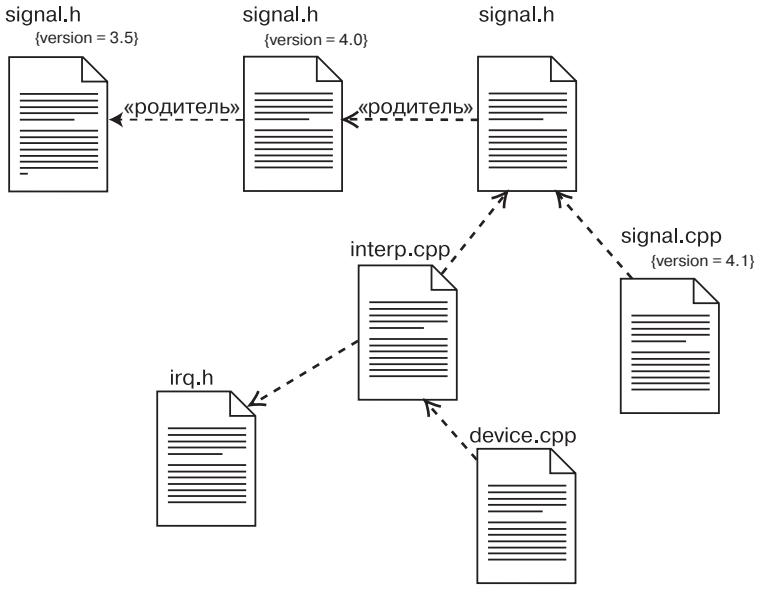


Рис. 30.2. Моделирование исходного кода

Подобные диаграммы легко генерируются путем обратного проектирования на основе информации, хранящейся в системе управления конфигурацией среды разработки.

Моделирование исполняемой версии

Выпуск версий простого приложения не составляет труда – единственная исполняемая программа сбрасывается на диск, и пользователи просто запускают ее. Для подобных приложений диаграммы артефактов не требуются: там нечего визуализировать, специфицировать, конструировать и документировать.

Выпуск версий более сложных приложений все-таки предполагает трудозатраты, иногда немалые. Кроме главной исполняемой программы (обычно файла с расширением .exe), нужен ряд вспомогательных модулей (обычно это файлы с расширением .dll, если вы работаете в контексте COM+, либо .class или .jar – при работе с языком Java), баз данных, файлов подсказки и ресурсных файлов. Для распределенных систем, вероятно, понадобится несколько исполняемых программ и других файлов, разбросанных по разным узлам. Если вы работаете с системой приложений, одни артефакты могут оказаться уникальными, а другие будут использоваться в нескольких приложениях. По мере развития системы управление конфигурацией всего этого множества артефактов приобретает все

большую важность и вместе с тем усложняется, поскольку изменение некоторых артефактов в одном приложении может отразиться на работе других.

По этой причине для визуализации, специфицирования, конструирования и документирования исполняемых версий (включая артефакты размещения, формирующие каждую версию, и связи между этими артефактами) используются диаграммы артефактов. Вы можете применять их для прямого проектирования новой системы и для обратного проектирования существующей.

Создавая диаграммы артефактов, подобные той, что изображена на рис. 30.2, вы на самом деле просто моделируете часть сущностей и связей, которые представляют реализацию вашей системы. По этой причине каждая диаграмма артефактов должна сосредотачивать внимание только на одном текущем наборе артефактов.

Чтобы смоделировать исполняемую версию, необходимо:

Механизмы расширения UML обсуждаются в главе 6, интерфейсы – в главе 11.

- Идентифицировать набор артефактов, подлежащих моделированию. Обычно это выборка либо вся совокупность артефактов, размещенных на одном узле, или же ряд артефактов, распределенных по узлам системы.
- Рассмотреть стереотипы каждого артефакта в наборе. Для большинства систем обнаружится ограниченное число разных видов артефактов (таких как исполняемые программы, библиотеки, таблицы, файлы и документы). Для визуального представления этих стереотипов можно воспользоваться механизмами расширения UML.
- Для каждого артефакта в наборе рассмотреть его связь с окружающими его артефактами. Чаще всего это интерфейсы, которые экспортирует (реализует) один артефакт и импортируют (используют) другие. Если нужно раскрыть соединения в системе, – явно смоделировать эти интерфейсы. Если же необходимо представить модель на более высоком уровне абстракции, раскрыть эти связи, показывая только зависимости между артефактами.

В качестве примера на рис. 30.3 представлена модель части исполняемой версии автономного робота. Основное внимание обращено на артефакты размещения, ассоциированные с функциями перемещения робота и вычислительными операциями. Вы видите один артефакт – `driver.dll`, который материализует компонент `Driving` (Перемещение), экспортирующий интерфейс `IDrive` (), импортируемый компонентом `Path` (Путь), материализуемым другим артефактом, `path.dll`. Зависимость между компонентами `Path` и `Driving` указывает на зависимость между артефактами `path.dll` и `driver.dll`, которые реализуют их. На диаграмме присутствует еще

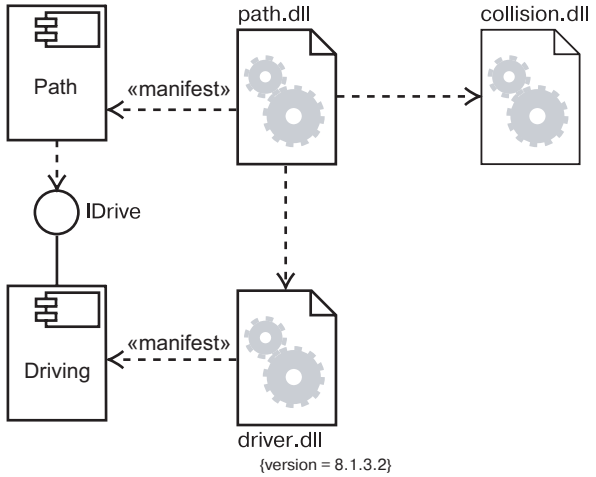


Рис. 30.3. Моделирование исполняемой версии

один артефакт – `collision.dll`, который также материализует компонент, хотя эти подробности и скрыты (показана лишь прямая зависимость `path.dll` от `collision.dll`).

В данной системе участвует множество других артефактов. Однако диаграмма сосредоточена на тех артефактах размещения, которые непосредственно относятся к процессу движения робота. Заметьте, что при такой компонентной архитектуре вы могли бы заменить конкретную версию `driver.dll` на другую – при условии, что она реализует те же (и, возможно, какие-то дополнительные) интерфейсы, – и `path.dll` мог бы по-прежнему правильно работать.

Моделирование физической базы данных

Моделирование схемы логической базы данных обсуждается в главе 8.

Логическая схема базы данных охватывает словарь хранимых данных системы вместе с семантикой их связей. Физически все эти сущности сохраняются в базе данных – либо в реляционной, либо в объектно-ориентированной, либо в гибридной (объектно-реляционной) – для последующего извлечения. UML так же хорошо приспособлен к моделированию физических баз данных, как и к моделированию их логических схем.

Отображение логической схемы базы данных на объектно-ориентированную базу достаточно прямолинейно, поскольку даже сложные цепочки наследования могут быть сохранены без какого-либо преобразования. Однако отобразить логическую схему на реляционную базу не так просто. Если имеет место наследование, приходится принимать решения относительно того, как отображать

Проектирование физических баз данных не входит в круг тем, рассматриваемых в этой книге, – здесь мы сосредоточиваем внимание на показе отдельных приемов моделирования баз данных и таблиц в UML.

классы на таблицы. Обычно применяется либо одна из трех ниженазванных стратегий, либо их комбинация:

1. *Спуск* (push down) – определяется отдельная таблица для каждого класса. Это простой, но примитивный подход, поскольку он вызывает проблемы сопровождения при добавлении новых дочерних классов или модификации существующих родительских;
2. *Подъем* (pull up) – все цепочки наследования свертываются так, что реализации любого класса в иерархии имеют одинаковое состояние. Недостаток в том, что во многих экземплярах приходится хранить избыточную информацию;
3. *Расщепление таблиц* (split tables) – данные родительского и дочернего класса разносятся по разным таблицам. Такой подход лучше отображает структуру наследования, но его недостаток состоит в том, что для доступа к данным приходится соединять многие таблицы.

Проектируя физическую базу данных, необходимо также решить, как следует отображать операции, определенные в логической схеме. Объектно-ориентированные базы данных обеспечивают достаточно прозрачное отображение, но что касается реляционных, здесь приходится решать, как реализовать эти логические операции. Опять же, существует несколько вариантов:

1. Простые операции создания, выборки, обновления и удаления реализуются стандартными вызовами SQL или ODBC;
2. Более сложное поведение (такое как бизнес-правила) отображается на триггеры и хранимые процедуры.

С учетом приведенных соображений для моделирования физической базы данных необходимо:

- ❑ Идентифицировать присутствующие в модели классы, которые представляют логическую схему базы данных.
- ❑ Выбрать стратегию отображения этих классов на таблицы. Следует рассмотреть и физическое распределение баз данных. Необходимо проанализировать физическое размещение данных в системе – от этого тоже зависит выбор стратегии.
- ❑ Создать диаграмму артефактов для визуализации, специфицирования, конструирования и документирования отображения, в которой артефакты имеют стереотип таблицы.
- ❑ По возможности использовать инструментальные средства для преобразования логической схемы в физическую.

На рис. 30.4 показан набор таблиц базы данных, взятых из информационной системы учебного заведения. Вы видите одну базу – `school.db`, изображенную в виде артефакта со стереотипом `database`.

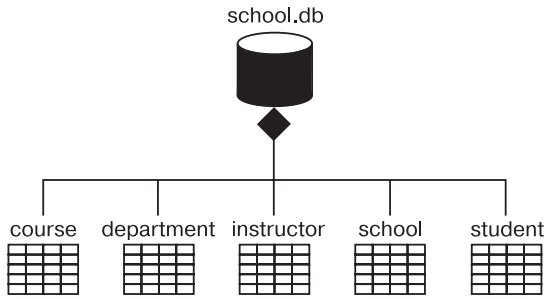


Рис. 30.4. Моделирование физической базы данных

Она состоит из пяти таблиц: `student` (студент), `class` (класс), `instructor` (инструктор), `department` (отдел) и `course` (курс). В соответствующей логической схеме базы данных нет наследования, поэтому отобразить ее на физическую несложно.

Хотя в данном примере это не показано, вы можете специфицировать содержимое каждой таблицы. Артефакты могут иметь атрибуты, поэтому общая идиома при моделировании физической базы данных заключается в использовании атрибутов для специфицирования столбцов каждой таблицы.

Аналогичным образом артефакты могут иметь операции, которые позволяют отображать хранимые процедуры.

Моделирование адаптируемых систем

Все диаграммы артефактов, показанные до сих пор, использовались для моделирования статических представлений. Их артефакты проводили всю свою жизнь на одном узле. Хотя эта ситуация встречается чаще всего, иногда, особенно при работе со сложными распределенными системами, все же приходится моделировать и динамические представления. Например, можно представить систему, которая реплицирует свои базы данных на несколько узлов, переключаясь на резервный сервер в случае отказа главного. Аналогично при моделировании глобальной распределенной системы, работающей в режиме 24x7 (то есть 7 дней в неделю, 24 часа в сутки), вы, скорее всего, столкнетесь с мобильными агентами – артефактами, которые мигрируют с одного узла на другой для обслуживания некоторой транзакции. Чтобы смоделировать такие динамические представления, вам понадобится использовать комбинацию диаграмм артефактов, объектов и взаимодействия.

Для моделирования адаптируемой системы необходимо:

- Рассмотреть физическое распределение артефактов, которые могут мигрировать с одного узла на другой. Вы можете специфицировать местоположение экземпляра артефакта, пометив его атрибутом `location`, который можно отобразить на диаграмме артефактов.
- Если нужно смоделировать действия, вызывающие миграцию артефакта, создать с этой целью соответствующую диаграмму взаимодействия, которая будет содержать экземпляры артефактов. Изменение местоположения артефакта можно проиллюстрировать, нарисовав один экземпляр несколько раз, но с разными значениями состояния, включая его местоположение.

Атрибут `location` обсуждается в главе 24, диаграммы объектов – в главе 14.

На рис. 30.5 представлено моделирование репликации базы данных, уже рассматривавшейся выше (см. рис. 30.4). Здесь мы видим два экземпляра артефакта `school.db`. Оба анонимны и имеют разные помеченные значения местоположения (`location`). Также присутствует примечание, которое явно указывает, какой экземпляр куда реплицируется.

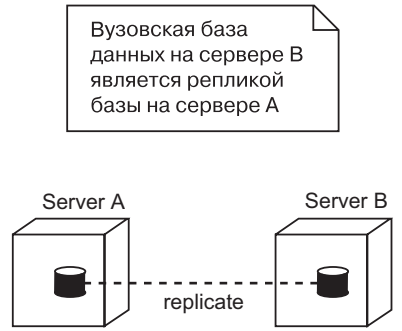


Рис. 30.5. Моделирование адаптируемых систем

Если необходимо показать подробности каждой базы данных, их можно изобразить в канонической форме – в виде артефакта со стереотипом `database`.

Хотя на этом рисунке это не показано, можно использовать и диаграмму взаимодействия для моделирования динамики переключения с главной базы данных на резервную.

Диаграммы взаимодействий обсуждаются в главе 19.

Прямое и обратное проектирование

Прямое и обратное проектирование артефактов довольно просто, потому что артефакты сами по себе – физические сущности (исполняемые программы, библиотеки, таблицы, файлы и разнообразные документы), а потому близки к работающей системе. При прямом проектировании класса или кооперации вы на самом

деле проектируете артефакт, который представляет исходный код, двоичную библиотеку или исполняемую программу для этого класса или кооперации. Точно так же при обратном проектировании исходного кода, двоичной библиотеки или исполняемой программы вы на самом деле выполняете обратное проектирование артефакта или набора артефактов, которые, в свою очередь, отображаются на классы или кооперации.

Приступая к прямому проектированию (созданию кода из модели) класса или кооперации, вы должны решить, во что нужно их преобразовывать: в исходный код, двоичную библиотеку или исполняемую программу. Логическую модель имеет смысл превратить в исходный код, если вас интересует управление конфигурацией файлов, которые затем будут переданы среде разработки. Если же вас интересует управление артефактами, которые фактически будут развернуты в составе работающей системы, то логические модели следует непосредственно преобразовать в двоичные библиотеки или исполняемые программы. Иногда нужно и то, и другое. Классу или кооперации можно поставить в соответствие как исходный код, так и двоичную библиотеку или исполняемую программу.

Чтобы выполнить прямое проектирование диаграммы артефактов, необходимо:

- Идентифицировать классы и кооперации, которые реализует каждый артефакт. Выразить эту реализацию при помощи связи материализации.
- Выбрать форму представления каждого артефакта. Это может быть либо исходный код (форма, поддающаяся управлению средствами разработки), либо двоичная библиотека или исполняемая программа (форма, которая может быть включена в работающую систему).
- Использовать инструментальные средства для прямого проектирования модели.

Обратное проектирование диаграмм классов обсуждается в главе 8.

Обратное проектирование диаграммы артефактов (создание модели из кода) – не идеальный процесс, поскольку при этом всегда происходит потеря информации. По исходному коду можно обратно спроектировать классы – это в общем-то тривиальная задача. Обратное проектирование артефактов из исходного кода выявляет существующие между файлами зависимости компиляции. Если же говорить о двоичных библиотеках, то самое большее, на что можно рассчитывать, – обозначить библиотеку как артефакт, а затем, используя обратное проектирование, раскрыть его интерфейсы. Это второе по распространенности действие, которое выполняется над диаграммами артефактов. Такой подход может быть полезен при ознакомлении с новыми плохо документированными

библиотеками. Максимум, что можно сделать в отношении исполняемой программы, – обозначить ее как артефакт, а затем дизассемблировать ее код, но вряд ли стоит этим заниматься, если только вы не пишете на языке ассемблера.

Чтобы выполнить обратное проектирование диаграммы артефактов, необходимо:

- Выбрать целевое представление. Исходный код можно реконструировать в артефакты, а затем в классы. Двоичные библиотеки можно подвергнуть обратному проектированию, чтобы раскрыть их интерфейсы. Исполняемые программы поддаются обратному проектированию в наименьшей степени.
- Используя инструментальные средства, указать код, который следует подвергнуть обратному проектированию. Использовать инструменты для генерации новой модели или модификации существующей, для которой ранее было выполнено прямое проектирование.
- Опрашивая модель и используя инструментальное средство, создать диаграмму артефактов – например, начать с одного или нескольких артефактов, а затем расширить диаграмму, следуя по связям или переходя к соседним артефактам. Отображать или скрывать детали этой диаграммы артефактов в соответствии с вашими установками.

В качестве примера на рис. 30.6 показана диаграмма артефактов, которая представляет результат обратного проектирования артефакта ActiveX `vbrun.dll`. Видно, что он реализует 11 интерфейсов.

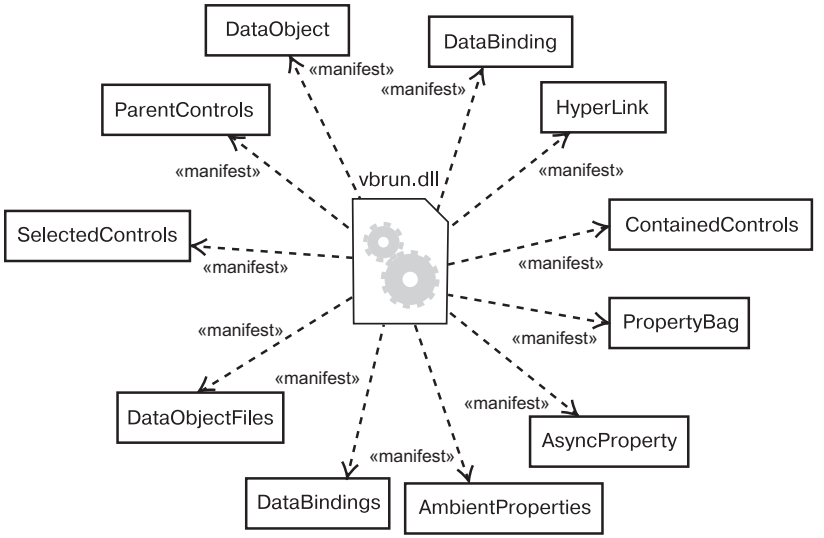


Рис. 30.6. Обратное проектирование диаграммы артефактов

Имея перед глазами такую диаграмму, можно понять семантику артефакта, перейдя к раскрытию его интерфейсных классов.

Чаще всего при обратном проектировании исходного кода, а иногда и двоичных библиотек и исполняемых программ, прибегают к помощи системы управления конфигурацией. Это означает, что вы будете работать с конкретными версиями файлов и библиотек, совместимых друг с другом. В таких случаях полезно включать помеченное значение, указывающее версию артефакта, – ее может предоставить система управления конфигурацией. Таким образом, можно использовать UML для визуализации истории артефакта при смене версий системы.

Советы и подсказки

При создании диаграмм артефактов на UML следует помнить о том, что каждая такая диаграмма – это графическое представление статического представления реализации системы. А это значит, что ни одна отдельная диаграмма артефактов не должна отражать все, что известно о представлении реализации системы. Все диаграммы артефактов, взятые в совокупности, представляют систему с точки зрения реализации, но каждая в отдельности описывает лишь один аспект.

Хорошо структурированная диаграмма артефактов обладает следующими свойствами:

- сосредоточена на выражении какого-то одного аспекта статического представления реализации системы;
- содержит только те элементы, которые существенны для понимания данного аспекта;
- представляет уровень детализации, соответствующий ее уровню абстракции, включая лишь те дополнения, которые важны для понимания на этом уровне;
- не настолько лаконична, чтобы неверно информировать читателя о важной семантике.

Изображая диаграмму артефактов, необходимо руководствоваться следующими принципами:

- присваивать диаграмме имя, описывающее ее назначение;
- располагать элементы так, чтобы минимизировать количество пересекающихся линий;
- пространственно организовывать элементы так, чтобы семантически близкие сущности оказывались рядом;
- использовать примечания и цветовые выделения для привлечения внимания к важным моментам диаграммы;
- осторожно применять элементы со стереотипами. Выделять небольшой набор общих для проекта (или организации) пиктограмм и использовать их согласованно.

Глава 31. Диаграммы размещения

В этой главе

- **Моделирование встроенных систем**
- **Моделирование клиент-серверных систем**
- **Моделирование полностью распределенных систем**
- **Прямое и обратное проектирование**

Диаграммы артефактов – второй вид диаграмм для моделирования физических аспектов объектно-ориентированных систем – обсуждаются в главе 30.

Диаграммы размещения – это один из двух видов диаграмм, используемых при моделировании физических аспектов объектно-ориентированной системы. Такая диаграмма представляет конфигурацию узлов, где производится обработка информации, и показывает, какие артефакты размещены на каждом узле.

Диаграммы размещения используются для моделирования статического представления системы с точки зрения размещения. В основном под этим понимается моделирование топологии аппаратных средств, на которых работает система. По существу, диаграммы размещения – это просто диаграммы классов, сосредоточенные на системных узлах.

Диаграммы размещения важны не только для визуализации, специфицирования и документирования встроенных, клиент-серверных и распределенных систем, но и для управления исполняемыми системами с использованием прямого и обратного проектирования.

Введение

При создании программной системы вы как разработчик программного обеспечения обращаете внимание в первую очередь на архитектуру и размещение своих программ в вычислительной среде. Но в качестве системного инженера вы заинтересованы главным образом в аппаратных и программных средствах системы и в том, как достичь оптимального их сочетания. Иными словами, разработчики программного обеспечения имеют дело с неосязаемыми артефактами вроде модели и кода, а разработчики систем – еще и с аппаратурой, вполне осязаемой.

Хотя основное назначение языка UML – визуализация, специфицирование, конструирование и документирование программных артефактов, он применим также и для работы с аппаратными артефактами. Из этого не следует, что UML – универсальный язык описания аппаратных средств наподобие VHDL. Однако он все же способен моделировать многие аппаратные аспекты системы, чего разработчику программного обеспечения достаточно для описания платформы, на которой система будет работать, а системному инженеру – для сопряжения программных и аппаратных средств. В UML представление о структуре программной системы дают диаграммы классов и компонентов. Для специфицирования поведения программ применяются диаграммы последовательности, коммуникации, состояния и деятельности. А на стыке программной и аппаратурной сфер находятся диаграммы размещения, позволяющие говорить о топологии процессоров и устройств, на которых выполняется система.

В UML диаграммы размещения используются для визуализации статических аспектов физических узлов и их взаимосвязей, а также для описания их деталей, которые имеют отношения к конструированию систем (см. рис. 31.1).

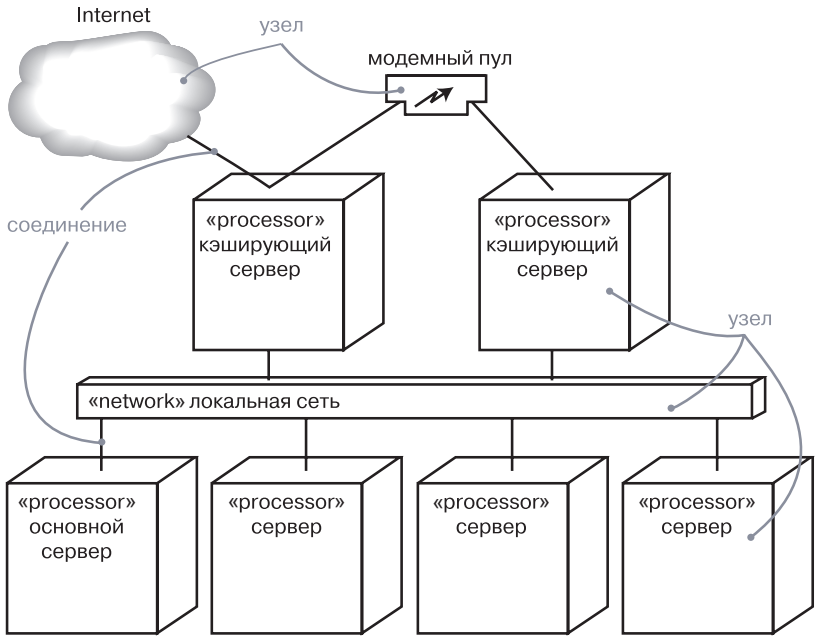


Рис. 31.1. Диаграмма размещения

Базовые понятия

На *диаграмме размещения* (deployment diagram) показана конфигурация обрабатывающих узлов и артефактов, размещенных в этих узлах. Диаграмма размещения представлена в виде графа с ребрами и вершинами.

Общие свойства

Общие свойства диаграмм обсуждаются в главе 7.

Диаграмма размещения обладает общими свойствами, присущими всем диаграммам, – именем и графическим содержанием, которое отражает одну из проекций модели. От других диаграмм отличается своим специфическим наполнением.

Содержание

Узлы обсуждаются в главе 27, связи – в главах 5 и 10, артефакты – в главе 26, пакеты – в главе12, подсистемы – в главе 32, экземпляры – в главе 13, диаграммы – в главе 8.

Диаграммы размещения обычно включают в себя узлы, а также связи зависимости и ассоциации; подобно всем прочим диаграммам, могут содержать примечания и ограничения. На диаграммах размещения бывают представлены артефакты, каждый из которых должен располагаться на каком-нибудь узле, а кроме того, пакеты или подсистемы, – и те и другие используются для группирования элементов модели в крупные блоки. Иногда бывает полезно поместить в диаграмму объектов еще и экземпляры, особенно если вы хотите визуализировать один экземпляр из семейства топологии аппаратных средств.

На заметку. Во многих отношениях диаграмма размещения является разновидностью диаграмм классов, фокусирующей внимание прежде всего на системных узлах.

Типичное применение

Диаграммы размещения используются для моделирования статического вида системы с точки зрения размещения. Это представление в первую очередь обращено на распределение, поставку и установку частей, из которых состоит физическая система.

Есть несколько типов систем, для которых диаграммы размещения не нужны. Если вы разрабатываете программу, исполняемую на одной машине и обращающуюся только к стандартным устройствам на этой же машине, управление которыми полностью возложено на операционную систему (возьмем для примера клавиатуру, дисплей и модем персонального компьютера), то диаграммы размещения можно игнорировать. Но если разрабатываемая программа

обращается к устройствам, которыми операционная система обычно не управляет, или эта программа физически размещена на разных процессорах, то диаграмма размещения поможет выявить отношения между программными и аппаратными средствами.

При моделировании статического представления системы с точки зрения размещения диаграммы размещения используются, как правило, в трех случаях:

1. *Для моделирования встроенных систем.* Встроенной (embedded) системой называется аппаратный комплекс, взаимодействующий с физическим миром, в котором велика роль программного обеспечения. Встроенные системы управляют двигателями, приводами и дисплеями, а сами управляются внешними воздействиями, например датчиками температуры и перемещения. Диаграмму размещения можно использовать для моделирования устройств и процессоров, из которых состоит встроенная система.
2. *Моделирование клиент-серверных систем.* Клиент-серверная (client/server) система – это типичный пример архитектуры, где основное внимание уделяется четкому распределению обязанностей между интерфейсом пользователя, существующим на клиенте, и хранимыми данными системы, существующими на сервере. Клиент-серверные системы находятся на одном конце спектра распределенных систем и требуют от вас принятия решений о том, как связать клиенты и серверы сетью, а также о том, как физически распределены программные артефакты между узлами. Диаграммы размещения позволяют моделировать топологию такой системы.
3. *Моделирование полностью распределенных систем.* На другом конце спектра распределенных систем находятся те из них, которые распределены широко или даже глобально (fully distributed) и охватывают серверы различных уровней. Часто на таких системах устанавливаются разные версии программных компонентов, часть которых даже мигрирует с одного узла на другой. Проектирование подобной системы требует решений, которые допускают непрерывное изменение системной топологии. Диаграммы размещения можно использовать для визуализации текущей топологии и распределения артефактов системы, чтобы можно было осмысленно говорить о влиянии на нее различных изменений.

Типичные приемы моделирования

Моделирование встроенной системы

Устройства и узлы обсуждаются в главе 27.

Разработка встроенной системы не сводится к созданию программного обеспечения: ведь приходится управлять физическим миром, где движущиеся части имеют склонность ломаться, сигналы зашумлены, а поведение нелинейно. При моделировании такой системы нужно принимать во внимание взаимодействие ее интерфейса с внешним миром, а это, как правило, нестандартные устройства и узлы.

Механизмы расширения UML обсуждаются в главе 6.

Диаграммы размещения способны облегчить общение инженеров-электронщиков и разработчиков программного обеспечения. Используя узлы со стереотипами, пиктограммы которых напоминают знакомые устройства, можно создавать диаграммы, которые одинаково понятны и тем, и другим. Диаграммы размещения также помогают анализировать баланс между программными и аппаратными средствами. Таким образом, они применяются для визуализации, специфицирования, конструирования и документирования проектных решений в части системной инженерии.

Для моделирования встроенной системы следует:

- Идентифицировать уникальные устройства и узлы.
- Позаботиться о визуальных обозначениях нестандартных устройств, воспользовавшись механизмами расширения UML для присвоения специфическим стереотипам подходящих пиктограмм. Как минимум необходимо различать процессоры, на которых размещены программные компоненты, и устройства, которые на данном уровне абстракции не содержат таковых.
- Смоделировать отношения между процессорами и устройствами на диаграмме размещения. Кроме того, специфицировать связи между артефактами представления системы с точки зрения реализации и узлами представления системы с точки зрения размещения.
- При необходимости раскрыть описание наиболее «интеллектуальных» устройств, смоделировав их структуру на более подробной диаграмме размещения.

В качестве примера на рис. 31.2 показана аппаратная реализация автономного робота. Вы видите один узел – Pentium motherboard

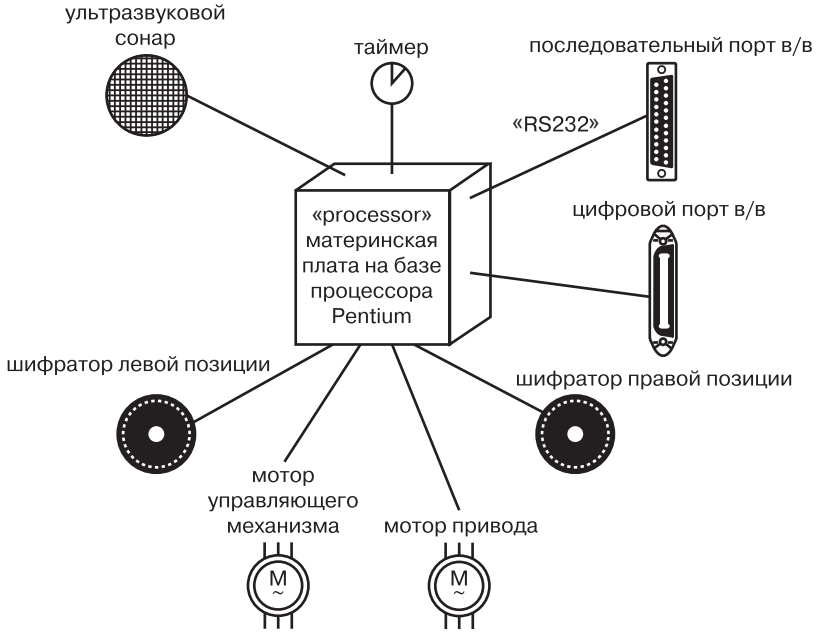


Рис. 31.2. Моделирование встроенной системы

(материнская плата на базе процессора Pentium) со стереотипом `processor` (процессор). Вокруг этого узла расположено восемь устройств, каждое из которых имеет стереотип `device` и представлено пиктограммой, похожей на реальный прототип.

Моделирование клиент-серверной системы

Приступая к разработке системы, программное обеспечение которой уже не помещается на одном процессоре, вы сразу же сталкиваетесь с целым рядом вопросов. Как оптимально распределить программные артефакты по узлам? Как они будут обмениваться информацией? Как быть с отказами и помехами? На одном конце спектра распределенных систем находятся клиент-серверные системы, в которых ясно прослеживается граница между пользовательским интерфейсом, обычно расположенном на клиенте, и данными, обычно хранящимися на сервере.

Здесь существует множество вариаций. Например, можно выбрать «тонкого» клиента, вычислительные мощности которого ограничены и который, следовательно, занят в основном взаимодействием с пользователем и отображением информации. У «тонких» клиентов может даже не быть собственных артефактов – вместо этого они загружают артефакты с сервера по мере необходимости, как, скажем, в случае с Enterprise JavaBeans. С другой стороны,

можно выбрать и «толстого» клиента, у которого вычислительных ресурсов больше и который вследствие этого может заниматься не только визуализацией. Выбор между «толстым» и «тонким» клиентом – это архитектурное решение, на которое влияют различные технические, экономические и политические факторы.

В любом случае при разделении системы на клиентскую и серверную части предстоит принять непростые решения о том, где физически разместить артефакты и как обеспечить баланс обязанностей между ними. Например, архитектура большинства систем управления информацией трехуровневая, то есть пользовательский интерфейс, бизнес-логика и база данных физически отделены друг от друга. Решение о том, куда поместить интерфейс и базу данных, как правило, очевидно, а вот понять, где должны находиться артефакты, реализующие бизнес-логику, куда сложнее.

Диаграммы размещения UML можно использовать для визуализации, специфицирования и документирования решений относительно топологии клиент-серверной системы и распределения программных артефактов между клиентом и сервером. Обычно требуется создать одну диаграмму размещения для системы в целом и ряд дополнительных диаграмм, детализирующих отдельные ее сегменты.

Для моделирования клиент-серверной системы вам понадобится:

- Идентифицировать узлы, представляющие процессоры клиента и сервера.
- Выделить те устройства, которые так или иначе влияют на поведение системы. Например, вы, скорее всего, захотите смоделировать устройство считывания кредитных карт и устройство отображения, отличные от стандартных мониторов, поскольку их расположение в топологии аппаратных средств системы имеет важное значение с точки зрения системной архитектуры.
- С помощью стереотипов разработать визуальные обозначения для процессоров и устройств.
- Смоделировать топологию узлов на диаграмме размещения. Также специфицировать связи между артефактами представления системы с точки зрения реализации и узлами представления системы с точки зрения размещения.

Пакеты обсуждаются в главе 12, множественность рассматривается в главе 10.

На рис. 31.3 показана топология системы, следующей классической клиент-серверной архитектуре. Мы видим, что граница между клиентом и сервером проведена явно путем использования пакетов `client` и `server`. Пакет `client` содержит два узла: `console` (консоль) и `kiosk` (киоск), имеющих свои стереотипы и потому визуально легко отличимых. Пакет `server` содержит два вида узлов: `caching server` (кэширующий сервер) и `server` (сервер), для каждого из которых

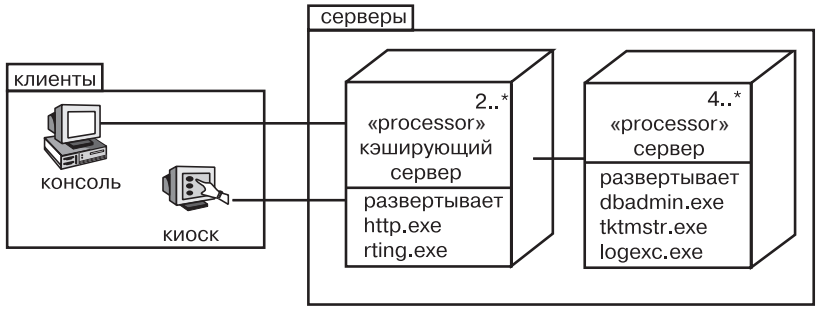


Рис. 31.3. Моделирование клиент-серверной системы

дополнительно описаны артефакты, размещаемые в узле. Заметьте, что как для `caching server`, так и для `server` указаны множественности, специфицирующие, сколько экземпляров ожидается в конкретной конфигурации. К примеру, из диаграммы видно, что кэширующих серверов должно быть не меньше двух.

Моделирование полностью распределенной системы

Распределенные системы могут быть самыми разными – от простых двухпроцессорных до разветвленных, размещенных на многих географически удаленных узлах. Последние, как правило, не бывают статическими. Узлы появляются и исчезают по мере изменения сетевого трафика и выходов процессоров из строя; создаются новые, более быстрые каналы связи, функционирующие параллельно медленным, постепенно устаревающим, которые в конце концов демонтируются. Изменяется не только топология системы, но и распределение программных артефактов. Например, таблицы баз данных могут реплицироваться между серверами с целью приблизить их к потребителю информации по мере изменения трафика. В некоторых глобальных системах артефакты могут мигрировать вслед за солнцем, перемещаясь с одного сервера на другой по мере того, как рабочий день начинается в одной части света и заканчивается в другой.

Визуализация, специфицирование и документирование топологии полностью распределенных систем представляют собой ценное подспорье для администратора, который должен вести учет вычислительных средств системы. Для этого можно применять диаграммы размещения UML. Документируя полностью распределенную систему, вы можете раскрыть детали сетевых устройств, представляя их в виде узлов со стереотипами.

Для моделирования полностью распределенной системы вам потребуется:

- Идентифицировать устройства и процессоры, как и в случае с более простой клиент-серверной системой.
- Если необходимо сделать выводы о производительности сетевой инфраструктуры или о влиянии на сеть изменений, – не забыть промоделировать коммуникационные устройства со степенью детализации, достаточной для такого рода оценок.
- Обратить особое внимание на логическое группирование узлов; для этого можно воспользоваться пакетами.
- Смоделировать устройства и процессоры с помощью диаграмм размещения. Всюду, где есть возможность, пользоваться инструментальными средствами, которые извлекают сведения о топологии системы путем обследования сети.
- Если необходимо сфокусировать внимание на динамике системы, – включить диаграммы вариантов использования для специфицирования поведения, представляющего интерес, и раскрыть варианты использования с помощью диаграмм взаимодействия.

Пакеты обсуждаются в главе 12.

Варианты использования обсуждаются в главе 17, диаграммы взаимодействия – в главе 19, экземпляры – в главе 13.

На заметку. При моделировании полностью распределенной системы саму сеть часто также изображают в виде узла. Например, можно представить Internet, как показано на рис. 31.1, в виде узла со стереотипом. Таким же образом позволяется оформить локальную (LAN) или глобальную (WAN) сеть – см. рис. 31.1. В любом случае вы можете воспользоваться атрибутами и операциями узла для описания свойств сети.

На рис. 31.4 показана топология полностью распределенной системы. Эта конкретная диаграмма размещения является также диаграммой объектов, поскольку содержит только экземпляры. Вы видите три консоли (анонимные экземпляры узла `console` со стереотипом), которые связаны с `Internet` (очевидно, узлом-одиночкой). С другой стороны, есть три экземпляра региональных серверов (`Regional server`), которые служат клиентской частью для национальных серверов (`Country server`), из которых показан только один. Как следует из примечания, национальные серверы соединены друг с другом, но такие связи на диаграмме не отражены.

На этой диаграмме вся сеть Internet представлена узлом со стереотипом.

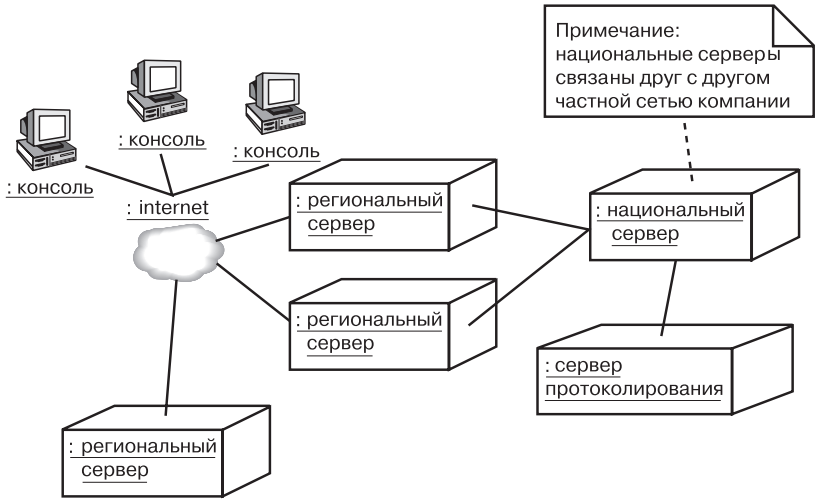


Рис. 31.4. Моделирование полностью распределенной системы

Прямое и обратное проектирование

Прямому проектированию (созданию кода по модели) диаграммы размещения поддаются лишь в очень небольшой степени. Например, после специфицирования физического распределения артефактов по узлам на диаграмме размещения можно воспользоваться инструментальными средствами, чтобы показать, как эти артефакты будут выглядеть в реальном мире. Для системных администраторов такое использование UML может оказать существенную помощь при решении весьма сложных задач.

Обратное проектирование (создание модели по коду) из реального мира в диаграммы размещения может иметь огромную ценность, особенно для полностью распределенных систем, которые постоянно изменяются. Чтобы приспособить UML к нужной предметной области, вы, вероятно, захотите подготовить набор узлов со стереотипами, понятных сетевому администратору. Преимущество использования UML состоит в том, что это стандартный язык, на котором можно выразить интересы не только администратора, но и разработчиков программного обеспечения.

Обратное проектирование диаграммы размещения следует осуществлять в таком порядке:

- Выбрать, что именно вы хотите подвергнуть обратному проектированию. В некоторых случаях вам нужно будет пройти по всей сети, но иногда достаточно ограничиться ее фрагментом.

- Выбрать степень детализации обратного проектирования. Зачастую работать можно на уровне всех процессоров системы, но бывает необходимо включить и сетевые периферийные устройства.
- Воспользоваться инструментальным средством, позволяющим совершить обход системы и показать ее топологию. Зафиксировать эту топологию в своей модели размещения.
- По ходу дела с помощью других инструментов выяснить, какие артефакты размещены в каждом узле, и занести их в модель размещения. Вам предстоит сложная поисковая работа, поскольку даже простейший персональный компьютер может содержать гигабайты разных артефактов, многие из которых не имеют отношения к вашей системе.
- Используя инструменты моделирования, создать диаграмму размещения путем опроса модели. Например, можно начать с визуализации базовой клиент-серверной топологии, а затем расширять диаграмму, добавляя в описания тех или иных узлов размещенные в них артефакты. Раскрыть детали содержания диаграммы размещения в той степени, в какой это требуется, чтобы донести ваши идеи до читателя.

Советы и подсказки

При создании в UML диаграмм размещения помните, что они являются всего лишь графическим представлением статического представления системы с точки зрения размещения. Это значит, что ни одна диаграмма, взятая сама по себе, не может описать все, что относится к размещению системы. Собранные вместе, диаграммы размещения дают полное представление о системе с соответствующей точки зрения; по отдельности же каждая диаграмма описывает лишь какой-то один аспект.

Хорошо структурированная диаграмма размещения обладает следующими свойствами:

- сосредоточена на каком-то одном аспекте статического представления системы с точки зрения размещения;
- содержит только те элементы, которые существенны для понимания данного аспекта;
- раскрывает только те детали, которые присутствуют на выбранном уровне абстракции, – показывает только те дополнения, которые существенны для понимания;
- не является настолько краткой, чтобы скрыть от читателя важную семантику.

Изображая диаграмму размещения, пользуйтесь следующими правилами:

- дайте диаграмме имя, соответствующее ее назначению;
- располагайте элементы так, чтобы число пересечений было минимальным и семантически близкие сущности оказывались рядом;
- используйте примечания и цветовые выделения, чтобы привлечь внимание к важным особенностям диаграммы;
- с осторожностью подходите к использованию элементов со стереотипами. Выберите ряд общих для вашего проекта или организации пиктограмм и применяйте их всюду и единообразно.

Глава 32. Системы и модели

В этой главе

UML – это графический язык для визуализации, специфицирования, конструирования и документирования артефактов программной системы. Его используют для того, чтобы моделировать системы. Модель – это упрощение реальности, абстракция, создаваемая, чтобы лучше понять систему. Система, зачастую разложенная на ряд подсистем, – это множество элементов, организованных некоторым образом для выполнения определенной цели. Система описывается набором моделей, по возможности рассматривающих ее с различных точек зрения. Важными составными частями модели являются такие сущности, как классы, интерфейсы, компоненты и узлы. В UML допускается моделирование систем и подсистем как единого целого – тем самым органично решается проблема масштабирования.

Хорошо структурированные модели помогают визуализировать, специфицировать, конструировать и документировать систему под разными (но вместе с тем взаимосвязанными) углами зрения. Хорошо структурированные системы функционально, логически и физически согласованы, но при этом состоят из слабо связанных друг с другом подсистем.

Введение

Для постройки собачьей будки серьезной подготовки не требуется. Потребности собаки просты, так что для их удовлетворения – если, конечно, не попался особо привередливый пес – вы просто сколачиваете будку без предварительного планирования.

Разница между постройкой собачьей будки и возведением небоскреба обсуждается в главе 1.

Возведение дома, а уж тем более небоскреба, требует больших затрат времени и сил. Потребности семьи или арендаторов офисного здания не столь примитивны, и для выполнения заказа даже самого покладистого клиента нельзя моментально начать строительство. Предварительно придется заняться моделированием. Различные участники процесса смотрят на него с разных точек зрения и преследуют разные интересы. Вот почему при строительстве сложных зданий предварительно создаются поэтажные планы, вертикальные разрезы, планы прокладки систем отопления и кондиционирования, планы электропроводки, водоснабжения и, возможно, даже прокладки вычислительных сетей. Разумеется, одной-единственной модели для адекватного учета всех этих аспектов недостаточно.

Диаграммы обсуждаются в главе 7, пять различных представлений архитектуры системы – в главе 2.

В UML все абстракции программной системы организуются в виде моделей, каждая из которых представляет относительно независимый, но важный аспект разрабатываемой системы. Для визуализации интересующих вас наборов этих абстракций можно использовать диаграммы. Рассмотрение пяти различных представлений архитектуры системы особенно полезно для удовлетворения потребностей различных участников процесса разработки программного обеспечения. В своей совокупности эти модели дают полное представление о структуре и поведении системы.

В больших системах множество элементов можно подвергнуть декомпозиции на более мелкие подсистемы, каждая из которых на более низком уровне абстракции может рассматриваться как отдельная система.

Механизмы расширения UML обсуждаются в главе 6, пакеты обсуждаются в главе 12.

В UML предусмотрены определенные средства для графического представления систем и подсистем (см. рис. 32.1). Эта нотация позволяет визуализировать декомпозицию системы на меньшие подсистемы. И система, и подсистема обозначаются пиктограммой компонента со стереотипом. Для моделей и представлений

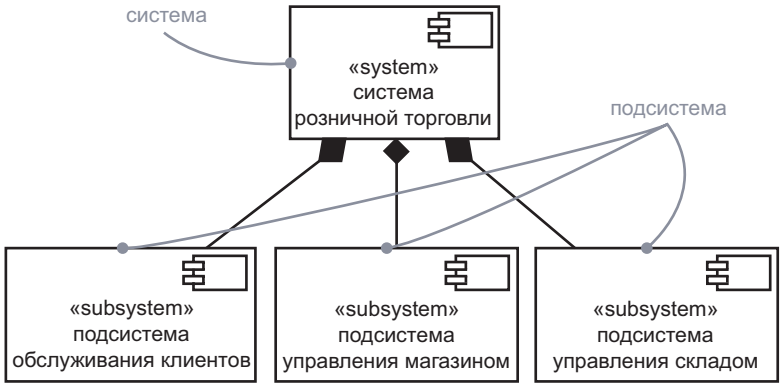


Рис. 32.1. Системы и подсистемы

предусмотрено особое графическое изображение (отличное от пакетов со стереотипом), хотя применяется оно редко, поскольку эти сущности в основном являются объектами манипуляции инструментальных средств, которыми пользуются для организации различных аспектов системы.

Термины и понятия

Система (system), возможно декомпозированная на ряд подсистем, – это множество элементов, организованных некоторым образом для выполнения определенной цели. Она описывается набором моделей, зачастую с различных точек зрения.

Подсистема (subsystem) – это объединение элементов, ряд которых составляет спецификацию поведения других ее элементов. Система и подсистема изображаются в виде пиктограммы компонента со стереотипом.

Модель (model) – это упрощение реальности, абстракция, создаваемая для лучшего восприятия системы.

Представление (view) – это проекция модели, рассматриваемая под определенным углом зрения: в ней отражены одни сущности и опущены другие, которые с данной точки зрения не представляют интереса.

Системы и подсистемы

Стереотипы обсуждаются в главе 6, пакеты – в главе 12, классы – в главе 4, варианты использования – в главе 17, автоматы – в главе 22, кооперации – в главе 28.

Система, собственно, и есть та сущность, которую вы разрабатываете и для которой строите модели. Система охватывает все артефакты, составляющие эту сущность, включая модели и элементы этих моделей (такие как классы, интерфейсы, компоненты, узлы и связи между ними). Все, что необходимо для визуализации, специфицирования, конструирования и документирования системы, является ее частью, а все, что для этой цели не требуется, лежит за пределами системы.

В UML система изображается в виде компонента со стереотипом, как показано на рис. 32.1. Являясь компонентом со стереотипом, она владеет некоторыми элементами. Если заглянуть внутрь системы, то можно увидеть все ее модели и отдельные элементы (в том числе и диаграммы), зачастую декомпозированные на более мелкие подсистемы. Являясь классификатором, система может иметь экземпляры (допускается существование нескольких экземпляров системы, размещенных в разных точках), атрибуты и операции (внешние по отношению к системе действующие лица способны воздействовать на систему в целом), варианты использования, автоматы и кооперации; все они могут принимать участие в специфицировании поведения системы. В ряде случаев она

обладает даже интерфейсами, что оказывается важным при конструировании системы систем.

Подсистема – это просто часть системы, которая используется для декомпозиции сложной системы на более простые, слабо зависящие друг от друга составляющие. То, что на одном уровне абстракции выглядит системой, на другом – более высоком – может рассматриваться как подсистема.

Агрегация и обобщение обсуждаются в главе 10.

Основная связь между системой и ее подсистемами – это композиция. Система (целое) может состоять из одной или нескольких подсистем (частей) либо вообще не содержать таковых. Допускается наличие связей обобщения между подсистемами. Благодаря этому можно моделировать семейства подсистем, ряд которых представляет общие виды систем, а другие являют собой специализацию этих систем, рассчитанную на конкретные условия. Подсистемы могут быть соединены друг с другом различными способами.

На заметку. Система – это сущность самого высокого уровня в данном контексте; составляющие ее подсистемы представляют полное разбиение системы на непересекающиеся части. Система – это подсистема верхнего уровня.

Модели и представления

Модель – это упрощение реального мира; реальность в ней описывается в контексте моделируемой системы. Проще говоря, модель – это абстракция системы. В то время как подсистема представляет собой разбиение множества элементов большей системы на независимые части, модель – это разбиение множества абстракций, используемых для визуализации, специфицирования, конструирования и документирования этой системы. Различие тонкое, но важное. Вы декомпозируете систему на подсистемы, чтобы их можно было разрабатывать и размещать в некоторой степени независимо друг от друга. Абстракции же системы или подсистемы вы разбиваете на модели, дабы лучше понять то, что собираетесь разрабатывать или размещать. Сложная система, например самолет, состоит из многих частей (каркас, реактивные двигатели, авиационная электроника, подсистема обслуживания пассажиров), причем эти подсистемы и система в целом могут моделироваться с разных точек зрения – в частности, с точки зрения конструкции, динамики, электросистемы, моделей отопления и кондиционирования.

Пакеты обсуждаются в главе 12.

Модель содержит набор пакетов. Явно представлять ее приходится не так уж часто. Однако инструментальные средства должны каким-то образом манипулировать моделями и обычно используют для их представления нотацию пакетов.

Пять различных представлений архитектуры системы обсуждаются в главе 2.

Модель владеет пакетами, которые, в свою очередь, состоят из неких элементов. Модели, ассоциированные с системой или подсистемой, исчерпывающим образом разбивают ее элементы. Это означает, что каждый элемент принадлежит одному и только одному пакету. Как правило, артефакты системы или подсистемы организуются в несколько неперекрывающихся моделей. Все возможные модели охватываются пятью представлениями архитектуры программного обеспечения.

Диаграммы обсуждаются в главе 7.

Модель (к примеру, процесса) может содержать так много артефактов – скажем, активных классов, связей и взаимодействий, – что в большой системе всю их совокупность нельзя охватить сразу. Представление системной архитектуры можно воспринимать как одну из проекций модели. Для каждой модели предусмотрен ряд диаграмм, с помощью которых удобно обозревать принадлежащие ей сущности. Представление охватывает подмножество сущностей, входящих в состав модели. Границы моделей представления обычно пересекать не могут. В следующем разделе будет показано, что между моделями нет прямых связей, хотя между элементами, содержащимися в различных моделях, могут существовать связи трассировки.

На заметку. UML не диктует вам, какими именно моделями следует пользоваться для визуализации, специфицирования, конструирования и документирования системы, хотя Rational Unified Process (см. приложение 2) предлагает некоторое множество моделей, проверенное на практике.

Трассировка

Связи обсуждаются в главе 5 и 10.

Специфицирование связей между такими элементами, как классы, интерфейсы, компоненты и узлы, – важная структурная составляющая модели. Для управления артефактами процесса разработки сложных систем, многие из которых существуют в нескольких версиях, большую роль играет определение связей между такими элементами, как документы, диаграммы и пакеты, присутствующие в разных моделях.

Зависимости обсуждаются в главе 5, стереотипы – в главе 6.

В UML концептуальные связи между элементами, содержащимися в разных моделях, можно представлять с помощью *трассировки* (trace relationship). К элементам в рамках одной модели трассировку применять нельзя. Изображается она в виде зависимости со стереотипом. Часто можно не обращать внимания на направление такой зависимости, хотя стрелка обычно указывает на элемент, возникший раньше по времени или более специфический

(см. рис. 32.2). Чаще всего связи трассировки используются, чтобы показать путь от требований до реализации, на котором лежат все промежуточные артефакты, а также для отслеживания версий.

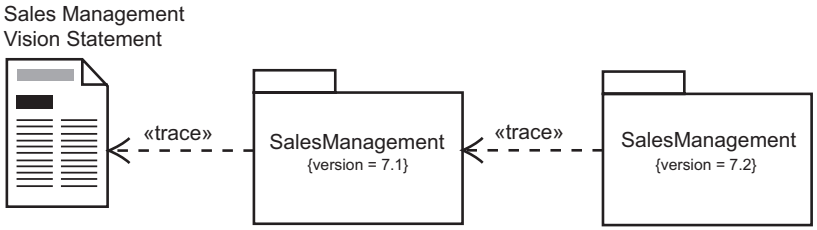


Рис. 32.2. Трассировка

На заметку. Как правило, вместо явного изображения связей трассировки имеет смысл пользоваться гиперссылками.

Типичные приемы моделирования

Моделирование архитектуры системы

Архитектура и моделирование обсуждаются в главе 1.

Наиболее распространенный случай применения систем и моделей – это организация элементов, используемых с целью визуализации, специфицирования, конструирования и документирования архитектуры системы. При этом затрагиваются все артефакты, встречающиеся в проекте разработки программного обеспечения. Моделируя системную архитектуру, вы сводите воедино решения, принятые относительно требований к системе, ее логических и физических элементов. Моделируются структурные и поведенческие аспекты системы, а также образцы, формирующие ее различные представления. Наконец, следует обратить внимание на соединения между подсистемами и трассировку решений, начиная от формулировки требований до размещения.

Для моделирования архитектуры системы необходимо:

Пять различных представлений архитектуры системы обсуждаются в главе 2, диаграммы – в главе 7.

- Идентифицировать представления, которые вы будете использовать для моделирования архитектуры. Чаще всего это представления с точки зрения вариантов использования, проектирования, взаимодействия, реализации и размещения, как показано на рис. 32.3.
- Специфицировать контекст системы, включая окружающие ее действующие лица.

При необходимости декомпозировать систему на элементарные подсистемы.

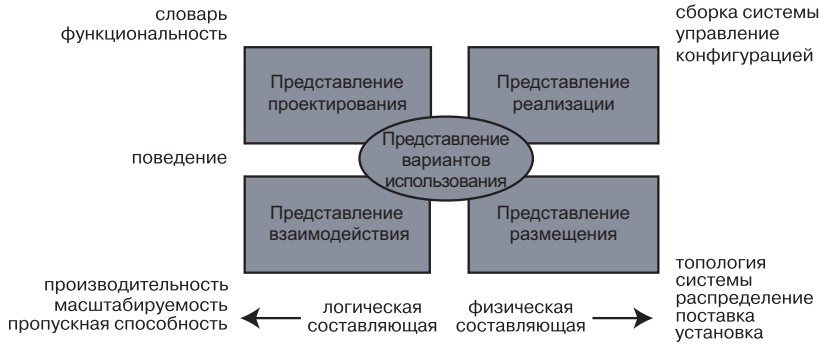


Рис. 32.3. Моделирование архитектуры системы

При моделировании системы в целом и ее подсистем вам понадобится:

- Специфицировать представление системы с точки зрения вариантов использования, которые описывают поведение системы, каким оно представляется конечным пользователям, аналитикам и тестировщикам. Для моделирования статических аспектов применить диаграммы вариантов использования, а для моделирования динамических – диаграммы взаимодействия, состояния и деятельности.
- Специфицировать представление системы с точки зрения проектирования, куда входят классы, интерфейсы и кооперации, формирующие словарь предметной области и предлагаемого решения. Для моделирования статических аспектов применить диаграммы классов и объектов, а для моделирования динамических – диаграммы взаимодействия, состояний и деятельности.
- Специфицировать представление системы с точки зрения взаимодействия, куда входят потоки, процессы и сообщения, формирующие механизмы параллелизма и синхронизации в системе. Использовать те же диаграммы, что и для представления проектирования, но основное внимание уделить активным классам и объектам, которыми представлены процессы и потоки, а также сообщениям и потоку управления.
- Специфицировать представление системы с точки зрения реализации, куда входят артефакты, используемые для сборки и выпуска физической системы. Для моделирования статических аспектов использовать диаграммы артефактов, для моделирования динамических – диаграммы взаимодействия, состояний и деятельности.
- Специфицировать представление системы с точки зрения размещения, куда входят узлы, формирующие топологию

аппаратных средств, на которых работает система. Для моделирования статических аспектов применить диаграммы размещения, а для моделирования динамических – диаграммы взаимодействия, состояний и деятельности.

- Смоделировать архитектурные образцы и образцы проектирования, формирующие перечисленные модели, с помощью кооперации.

Технология Rational Unified Process обсуждается в приложении 2.

Необходимо понимать, что системная архитектура не рождается сразу, под влиянием моментального озарения. Хорошо структурированный процесс применения UML подразумевает последовательное (итерационное и пошаговое) уточнение архитектуры на основе анализа вариантов использования.

Механизмы расширения UML обсуждаются в главе 6.

Если не брать в расчет простейшие системы, вам необходимо управлять версиями системных артефактов. Для представления решений о версиях каждого элемента можно воспользоваться механизмами расширения UML, в частности помеченными значениями.

Моделирование системы систем

То, что на одном уровне абстракции выглядит как система, на другом, более высоком, представляется подсистемой. Аналогичным образом то, что на одном уровне является подсистемой, вполне может рассматриваться как полноценная система группой разработчиков, ответственных за ее создание.

Такая иерархия наблюдается во всех сложных системах. По мере возрастания сложности системы вы встаете перед необходимостью ее декомпозиции на более простые, каждую из которых можно разрабатывать отдельно, а затем постепенно объединять их. Разработка подсистемы и системы подчиняется одним и тем же принципам.

Чтобы смоделировать систему систем, вам следует:

- Идентифицировать основные функциональные составляющие системы, которые можно разрабатывать, выпускать и размещать до некоторой степени независимо. На результаты этого разбиения системы часто влияют технические, политические и юридические факторы.
- Для каждой подсистемы специфицировать ее контекст так же, как это делается для системы в целом (притом в число действующих лиц, окружающих подсистему, включаются все соседние подсистемы, поэтому необходимо проектировать их совместную работу).
- Смоделировать архитектуру каждой подсистемы так же, как это делается для всей системы.

Советы и подсказки

Важно выбрать правильное множество моделей для визуализации, специфицирования, конструирования и документирования системы. Хорошо структурированная модель:

- дает упрощенное представление реальности с одной относительно независимой точки зрения;
- самодостаточна, то есть не требует для понимания ее семантики никакой дополнительной информации;
- слабо связана с другими моделями посредством связей трассировки;
- совместно с другими смежными моделями дает полное представление об артефактах системы.

Столь же важно бывает представить сложную систему в виде декомпозиции хорошо структурированных подсистем. Хорошо структурированная система:

- функционально, логически и физически согласована;
- может быть декомпозирована на почти независимые подсистемы, которые сами являются системами на более низком уровне абстракции;
- может быть визуализирована, специфицирована, сконструирована и документирована в виде набора взаимосвязанных, неперекрывающихся моделей.

Для моделей в UML предусмотрен специальный графический символ, однако лучше им не пользоваться; моделируйте только систему, а не саму модель. Инструментальные средства обычно располагают возможностями просмотра, организации и управления наборами моделей.

При изображении системы или подсистемы в UML:

- используйте каждую из них как начальную точку для всех артефактов, ассоциированных с ней;
- показывайте только основные виды агрегации между системой и ее подсистемами. Выносите детали связей между ними на диаграммы более низкого уровня.

Часть VII

Итоги

Глава 33. Применение UML

Глава 33. Применение UML

В этой главе:

Простые задачи моделируются с помощью UML без труда. Легко поддаются моделированию и сложные задачи, особенно если вы уже приобрели некоторый опыт использования этого языка.

Одного только чтения литературы о языке UML недостаточно, – чтобы в полной мере овладеть языком, необходимо применять его на практике. В зависимости от того, что вы уже знаете, начинать работу с UML можно по-разному. По мере приобретения опыта вы поймете и оцените более сложные его конструкции.

Если вы способны рассматривать и оценивать какой-либо предмет с разных позиций, то UML поможет его смоделировать.

Переход к UML

Согласно известному правилу 80/20, 80% большинства проблем можно смоделировать, используя 20% средств UML. Основных структурных сущностей – таких как классы, атрибуты, операции, варианты использования и пакеты, вместе с основными структурными связями – такими как зависимость, обобщение и ассоциация, вполне достаточно для создания статических моделей большинства типов проблемных областей. Добавьте к этому основные поведенческие сущности, например простые автоматы и взаимодействия, – и вы сможете моделировать множество полезных аспектов динамики систем. Необходимость в более развитых средствах UML вы почувствуете только тогда, когда столкнетесь с необходимостью моделирования более сложных вещей (скажем, параллельных и распределенных систем).

Концептуальные модели UML обсуждаются в главе 2.

Хорошей стартовой площадкой в использовании UML может быть моделирование некоторых базовых абстракций или поведения, которые уже присутствуют в ваших системах. Разработайте концептуальную модель UML, чтобы получить некоторый каркас, на основе которого вы будете расширять свое понимание языка.

Позднее вы станете лучше понимать, как работают совместно более развитые средства UML. По мере того как вы отважитесь на моделирование более сложных задач, погружайтесь в специфические средства UML, изучая общую технику моделирования по этой книге.

Если вы только начинаете осваивать объектно-ориентированные методы, воспользуйтесь следующими советами:

- Начните привыкать к идее абстракции. Коллективные упражнения с CRC-карточками и анализ вариантов использования – отличный способ повысить свое мастерство в идентификации сложных абстракций.
- Моделируйте простые статические части проблемы, используя классы, зависимости, обобщения и ассоциации с тем, чтобы лучше познакомиться с визуализацией сообществ абстракций.
- Используйте простые диаграммы последовательности и коммуникации, чтобы моделировать динамическую составляющую проблемы. Построение модели взаимодействия пользователя с системой – хорошая начальная позиция, которая принесет немедленную выгоду, помогая вам пройтись по наиболее важным вариантам использования системы.

Если вы новичок в моделировании:

- Для начала возьмите часть системы, которую вы уже построили (предпочтительно – на каком-либо объектно-ориентированном языке программирования, таком как C++), и создайте модель UML этих классов и их связей.
- Используя UML, попытайтесь выразить некоторые детали программных идиом или механизмов, которые вы использовали в системе и которые существуют в проекте, но еще не воплощены в коде.
- Попытайтесь реконструировать модель архитектуры приложения (особенно нетривиального), используя компоненты, в том числе подсистемы, для представления его основных структурных элементов. Для организации самой модели применяйте пакеты.
- После того как вы в целом освоите словарь UML, постройте модель той части системы, для которой вы собираетесь писать код. Продумайте структуру и поведение, которые вы специфицируете, и только потом, когда их размер, форма и семантика вас вполне устроят, используйте эту модель в качестве каркаса для реализации системы.

Если у вас уже есть опыт использования другого объектно-ориентированного метода:

- Просмотрите модель, построенную на привычном для вас языке моделирования, и отобразите ее элементы на элементы UML. Скорее всего, большинство из них окажутся идентичными, а потому изменения в массе своей будут носить косметический характер.
- Рассмотрите сложные задачи моделирования, с которыми вы сталкивались в своей практике и которые было невозможно решить с помощью привычного вам языка. Поищите какие-нибудь средства UML, способные упростить решение проблемы.

Если вы искушенный пользователь, вам пригодятся следующие рекомендации:

- Убедитесь, что вы постигли концептуальную модель UML. Вы можете упустить из виду гармоничность концепций этого языка, если сразу возьмете на вооружение самые сложные его средства, не изучив предварительно базовый словарь.
- Уделите особое внимание тем средствам UML, которые предназначены для моделирования внутренней структуры: кооперациям, параллелизму, распределению и образцам, то есть случаям применения сложной и тонкой семантики.
- Рассмотрите механизмы расширения UML и подумайте, как адаптировать язык, чтобы можно было напрямую выразить словарь вашей предметной области. Не поддавайтесь соблазну все усложнять, иначе созданную вами модель никто не поймет (разве что найдется чрезвычайно продвинутый пользователь).

Что дальше

Настоящее руководство пользователя – это всего лишь одна из широкого ряда книг, которые призваны помочь вам в изучении UML. В дополнение к ней мы можем порекомендовать следующую литературу:

- James Rumbaugh, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual. Second Edition. – Addison-Wesley, 2005. (Русскоязычное издание книги: Буч Г., Якобсон А., Рамбо Дж. UML. Классика CS. 2-е изд. – СПб: Питер, 2006.) Исчерпывающее руководство по синтаксису и семантике UML.
- Ivar Jacobson, Grady Booch, James Rumbaugh. The Unified Software Development Process. – Addison-Wesley, 1999. (Русскоязычное издание книги: Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. – СПб: Питер, 2002.) Советы по организации процесса разработки с применением UML.

Чтобы узнать больше о моделировании от самих авторов UML, воспользуйтесь следующими изданиями:

- Michael Blaha, James Rumbaugh. Object-Oriented Modeling and Design with UML. Second Edition. – Prentice Hall, 2005.
- Grady Booch. Object-Oriented Analysis and Design with Applications. Second Edition. – Addison-Wesley, 1993. (Русскоязычное издание книги: Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. 2-е изд. – М.: Издательство Бином; СПб.: Невский диалект, 1999.)
- Ivar Jacobson, Magnus Christerson, Patrik Jonsson, Gunnar Overgaard. Object-Oriented Software Engineering: A Use Case Driven Approach. – Addison-Wesley, 1992.

Информацию о технологии Rational Unified Process можно найти здесь:

- Philippe Kruchten. The Rational Unified Process: An Introduction. Third Edition. – Addison-Wesley, 2004.

Последнюю информацию об UML, равно как и самую последнюю версию стандарта UML, можно найти на Web-сайте OMG www.omg.org.

Существует немало другой литературы, где описываются UML и разнообразные методы разработки, а кроме того, огромное число книг, посвященное общей практике программной инженерии.

Приложение 1. Нотация UML

Общий обзор UML содержится в главе 2.

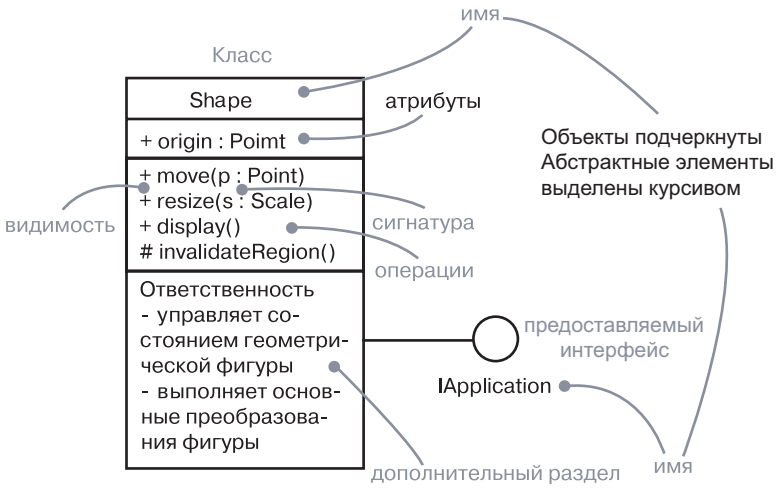
UML – это язык визуализации, специфицирования, конструирования и документирования артефактов программных систем. Как и любой язык, он характеризуется четко определенными синтаксисом и семантикой. Та часть синтаксиса UML, которую можно представить наглядно, выражена в его графической нотации.

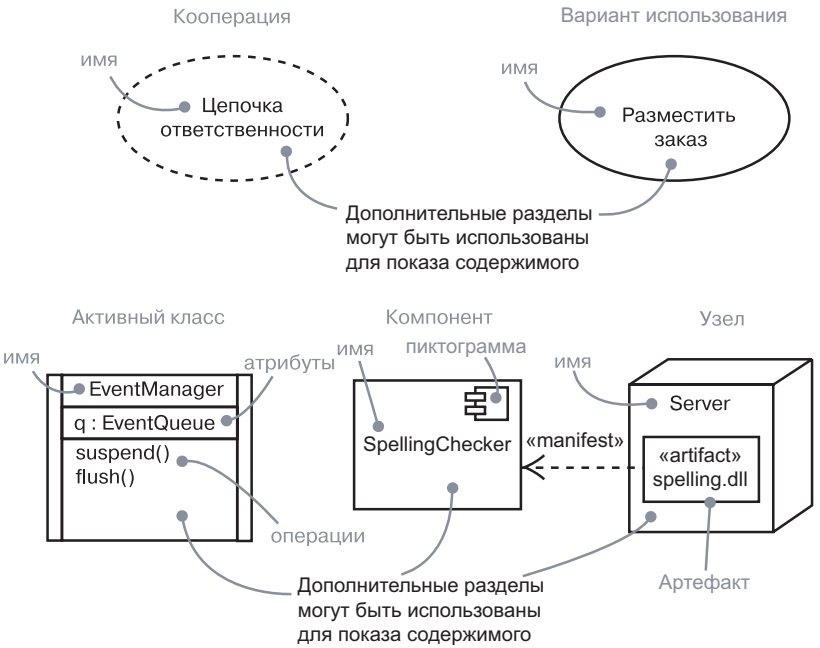
В этом приложении приводятся описания элементов нотации UML, рассматривавшихся выше.

Сущности

Структурные сущности

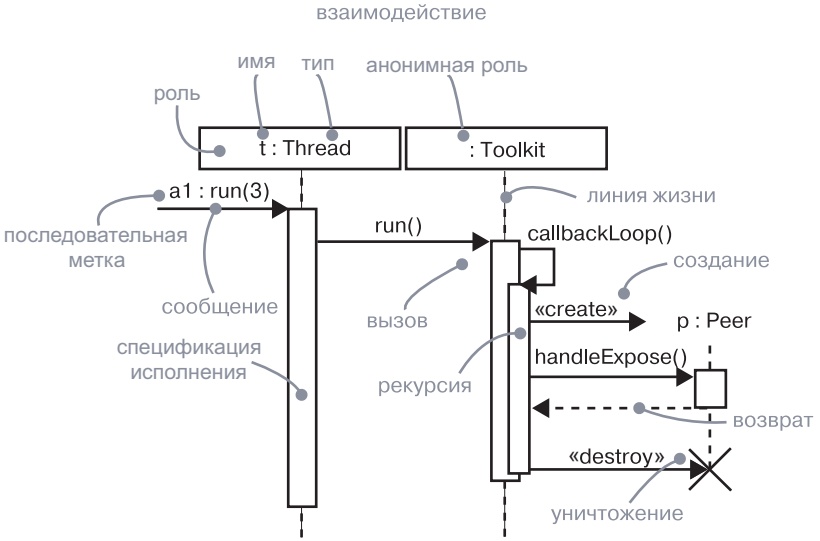
Структурные сущности – это «имена существительные» в UML-моделях. Они включают в себя классы, интерфейсы, кооперации, варианты использования, активные классы, компоненты и узлы.

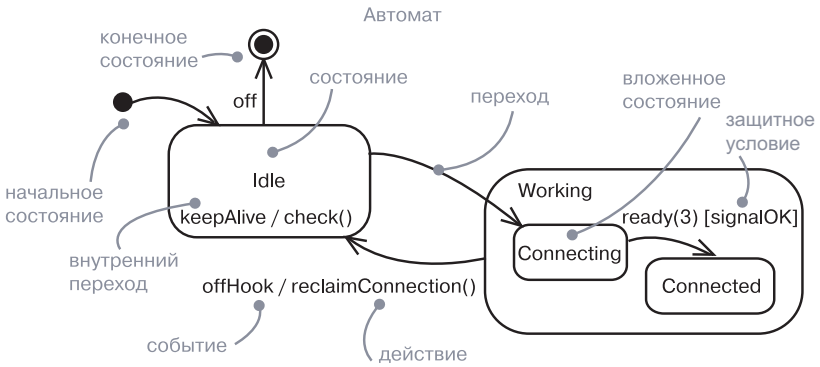




Поведенческие сущности

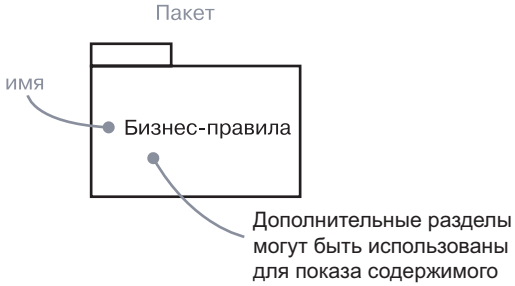
Поведенческие сущности – это динамические части UML-моделей. Они включают взаимодействия и автоматы.





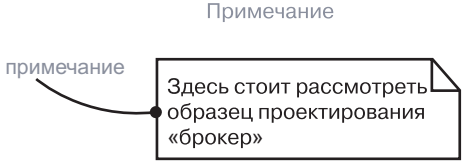
Группирующие сущности

Группирующие сущности – это организационные части моделей UML. К ним относятся пакеты.



Аннотирующие сущности

Аннотирующие сущности – это пояснительные части UML-моделей. К ним относятся примечания.



Связи

Зависимость

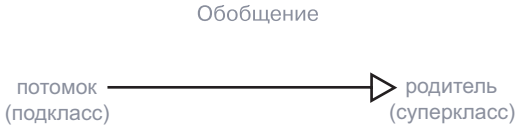
Зависимость – это семантическая связь между двумя сущностями, при которой изменение одной сущности (независимой) может повлиять на семантику другой (зависимой).

Ассоциация

Ассоциация – структурная связь, которая описывает набор ссылок. Ссылка – это экземпляр ассоциации, представляющий собой соединение объектов.

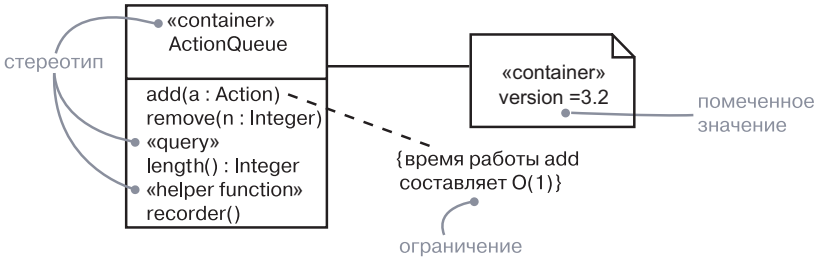
Обобщение

Обобщение – связь, при которой объекты специализированного элемента (потомка) могут замещать объекты более общего элемента (родителя).



Расширяемость

UML располагает тремя механизмами расширения синтаксиса и семантики языка: это стереотипы, которые представляют новые элементы модели, помеченные значения, представляющие ее новые атрибуты, и ограничения, представляющие ее новую семантику.



Диаграммы

Диаграмма – это графическое представление набора элементов, чаще всего отображаемых в виде связного графа с вершинами (сущностями) и ребрами (связями). Диаграмма является проекцией системы. В UML определено тринадцать видов диаграмм:

1. Диаграмма классов – структурная диаграмма, на которой показано множество классов, интерфейсов, кооперaций и связей между ними.
2. Диаграмма объектов – структурная диаграмма, показывающая множество объектов и связей между ними.
3. Диаграмма компонентов – структурная диаграмма, которая показывает внешние интерфейсы, включая порты, и внутреннюю композицию компонента.
4. Диаграмма составной структуры – структурная диаграмма, представляющая внешние интерфейсы и внутреннюю композицию структурированных классов. В этой книге диаграмма составной структуры рассматривается совместно с диаграммой компонентов ввиду того, что различия между ними весьма незначительны.
5. Диаграмма вариантов использования – поведенческая диаграмма, которая показывает набор вариантов использования, действующие лица и их связи.
6. Диаграмма последовательности – поведенческая диаграмма, которая показывает взаимодействия, подчеркивая временной порядок сообщений.

7. Диаграмма коммуникации – поведенческая диаграмма, которая представляет взаимодействие, подчеркивая структурную организацию объектов, посылающих и принимающих сообщения.
8. Диаграмма состояний – поведенческая диаграмма, которая показывает автомат (машину состояний), выделяя событийно упорядоченное поведение объекта.
9. Диаграмма деятельности – поведенческая диаграмма, которая показывает вычислительный процесс, выделяя поток управления от одной деятельности к другой.
10. Диаграмма размещения – структурная диаграмма, которая показывает связи множества узлов, артефактов, а также материализованных классов и компонентов. В этой книге мы также рассмотрели моделирование артефактов с привлечением диаграмм артефактов.
11. Диаграмма пакетов – структурная диаграмма, показывающая, как элементы модели организованы в пакеты.
12. Временная диаграмма – поведенческая диаграмма, отражающая взаимодействие с сообщениями в определенные моменты времени. В этой книге не рассматривается.
13. Диаграмма обзора взаимодействий – поведенческая диаграмма, сочетающая в себе аспекты диаграмм деятельности и последовательности. В этой книге не рассматривается.

Допускаются гибридные типы диаграмм; каждая из них может содержать элементы, которые чаще всего размещаются на диаграммах другого типа.

Приложение 2. Rational Unified Process

Процессом называется частично упорядоченное множество шагов, направленное на достижение некоторой цели. Если говорить о разработке программного обеспечения, то вашей целью является поставка в предсказуемые сроки продукта, удовлетворяющего потребностям бизнеса.

В значительной мере UML не зависит от процесса – в том смысле, что его можно использовать в различных процессах разработки программного обеспечения. Унифицированный процесс разработки программного обеспечения (Rational Unified Process, RUP) – это один из подходов к организации жизненного цикла программного обеспечения, который особенно хорошо сочетается с UML. Цель RUP – обеспечить создание программного продукта высочайшего качества, соответствующего потребностям пользователя, в заданные сроки и в пределах запланированного бюджета. RUP вобрал в себя лучшие из существующих методов разработки программного обеспечения и придал им форму, которая может быть легко адаптирована для самых разных проектов и организаций. С точки зрения управления проектом RUP предлагает упорядоченный подход к способам распределения заданий и обязанностей в организации, занимающейся разработкой программного обеспечения.

Настоящее приложение описывает основные элементы RUP.

Характеристики процесса

RUP – это *итерационный* процесс. Когда речь идет о простых системах, не представляет особого труда последовательно определить задачу, спроектировать ее целостное решение, построить программное обеспечение и затем протестировать конечный продукт. Но, учитывая сложность и разветвленность современных систем, такой линейный подход к разработке оказывается нереалистичным. Итерационный подход предполагает постепенное проникновение в суть проблемы путем последовательных уточнений и пошагового наращивания решения на протяжении нескольких циклов. Присущая итерационному процессу внутренняя гибкость позволяет включать в бизнес-цели новые требования и тактические изменения. Его применение

обеспечивает возможность выявлять и устранять риски, связанные с проектом, на возможно более ранних стадиях разработки.

Суть деятельности в рамках RUP – создание и сопровождение *моделей*, а не бумажных документов. Модели, особенно выраженные на UML, обеспечивают семантически богатое представление программной системы в процессе разработки. Их можно рассматривать с разных точек зрения; допускается электронный способ извлечения и контроля представленной в них информации. RUP сосредоточен на моделях, а не бумажных документах, с тем чтобы минимизировать накладные расходы, связанные с производством и сопровождением документов и повысить релевантность их информационного содержимого.

Разработка в рамках RUP *сконцентрирована на архитектуре*. Основное внимание уделяется ранней разработке и стабилизации архитектуры программного обеспечения. Наличие прочной архитектуры позволяет организовать параллельную разработку, свести к минимуму необходимость переделок, повысить степень повторного использования компонентов и в конечном счете облегчить последующее сопровождение. Этот архитектурный чертеж служит прочной основой для планирования и управления компонентной разработкой программного обеспечения.

Разработка в рамках RUP *управляется вариантами использования*. В основу построения системы положено исчерпывающее представление о том, каково ее назначение. Варианты использования и сценарии применяются на всех стадиях процесса – от формулирования требований до тестирования. Они помогают проследить все действия от начала разработки до поставки готового продукта.

RUP поддерживает *объектно-ориентированные методы*. В моделях RUP применяются понятия объектов и классов, а также связей между ними, и в качестве общей нотации используется нотация UML.

RUP – *конфигурируемый процесс*. Хотя ни один отдельно взятый процесс не в состоянии удовлетворить требованиям всех организаций, занимающихся разработкой программного обеспечения, RUP поддается настройке и масштабируется для использования как в совсем небольших коллективах разработчиков, так и в больших компаниях. Он базируется на простой и ясной архитектуре, которая обеспечивает концептуальное единство множества различных процессов разработки, но при этом адаптируется к различным ситуациям. Одна из составных частей RUP содержит рекомендации по его конфигурированию для нужд конкретной компании.

RUP обеспечивает постоянный *контроль качества* и *управление риском*. Контроль качества встроен в сам процесс и распространяется на все виды деятельности в рамках разработки и всех

ее участников; при этом используются объективные критерии и методы оценки. Этот аспект не рассматривается как автономный вид деятельности, который можно отложить до завершения работ. Управление риском также встроено в процесс и обеспечивает заблаговременное выявление и устранение препятствий на пути успешной реализации проекта.

Фазы и итерации

Фаза (phase) – это промежуток времени между двумя важными опорными точками процесса, в которых должны быть достигнуты четко определенные цели, подготовлены те или иные рабочие продукты и приняты решения о том, можно ли переходить к следующей фазе. Как показано на рис. П2.1, RUP состоит из следующих четырех фаз:

1. *Начальная фаза* (inception) – определение границ проекта, разработка концепции и начального плана проекта;
2. *Разработка* (elaboration) – проектирование, реализация и тестирование стабильной архитектуры, завершение разработки плана проекта;
3. *Конструирование* (construction) – построение первой эксплуатационной версии системы;

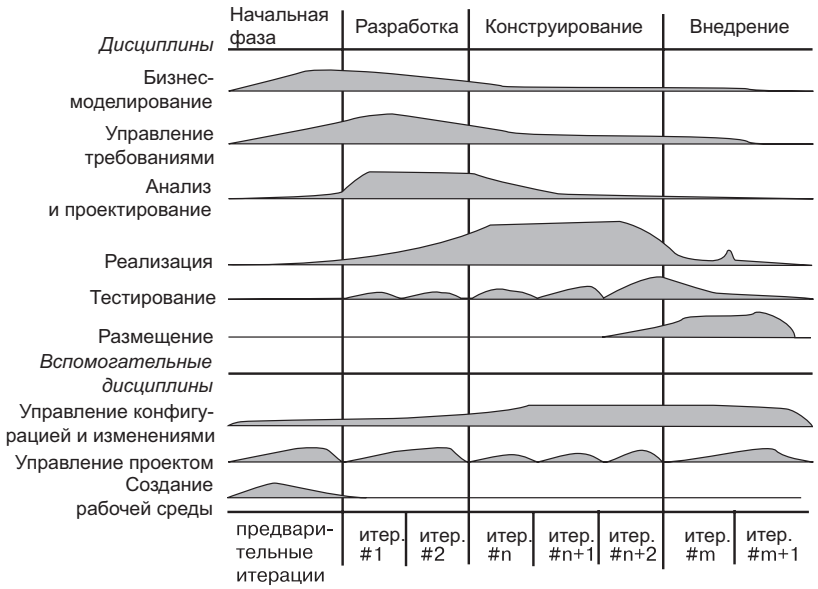


Рис. П2.1. Жизненный цикл процесса разработки программного обеспечения

4. *Внедрение* (transition) – поставка системы конечным пользователям.

Первая и вторая фазы сосредоточены главным образом на творческих и инженерных видах деятельности жизненного цикла разработки, в то время как третья и четвертая в большей степени являются производственными.

В пределах каждой фазы может происходить множество итераций. *Итерация* представляет полный цикл разработки, в результате которого появляется работающая версия, – от анализа и формулирования требований до реализации и тестирования. Такая версия не обязана включать весь набор возможностей коммерческого продукта. Ее назначение – обеспечить прочную базу для последующей оценки и тестирования, а также подвести промежуточный итог, необходимый для начала следующего цикла разработки.

Каждая фаза и итерация предполагает определенные трудозатраты, направленные на снижение риска. В конце каждой фазы определяется *опорная точка* (milestone), позволяющая оценить, в какой мере достигнуты намеченные цели и следует ли внести в процесс какие-либо изменения, прежде чем двигаться дальше.

Фазы

Начальная фаза

На этой стадии разрабатывается концепция системы, устанавливаются рамки проекта. В частности, определяются бизнес-цели проекта и высокоуровневые требования к нему, составляется начальный план. Пока что он включает в себя выработку критериев успеха, оценку риска, необходимые ресурсы и основные опорные точки. Нередко создается исполняемый прототип, подтверждающий реалистичность концепции. Обычно все перечисленные задачи решает небольшая группа людей.

В конце начальной фазы еще раз внимательно анализируется весь жизненный цикл проекта и принимается решение о том, стоит ли приступать к полномасштабной разработке.

Разработка

Цели этой фазы – анализ предметной области, выработка прочной архитектурной основы, уточнение плана проекта и исключение наиболее серьезных рисков. Архитектурные решения должны приниматься тогда, когда уже понятна вся система, то есть сформулирована большая часть требований. Для подтверждения правильности выбора архитектуры создается система, демонстрирующая

выбранные принципы в действии и реализующая некоторые наиболее важные варианты использования.

Основные действующие лица на данном этапе – архитектор системы и менеджер проекта, а также аналитики, разработчики, тестировщики и др. Обычно вторая фаза требует участия большего количества специалистов, чем первая.

В конце фазы разработки изучаются максимально конкретизированные цели проекта, его рамки, выбор архитектуры и методы разрешения основных рисков, а затем принимается решение о переходе к конструированию.

Конструирование

На стадии конструирования производится итерационная и пошаговая разработка продукта, готового к внедрению. Под этим подразумевается описание ранее не учтенных требований и критериев приемки, материализация дизайна, завершение реализации и тестирования программного обеспечения.

В такую работу вовлечены архитектор системы, менеджер проекта и лидеры групп разработчиков, а кроме того, весь персонал, занятый разработкой и тестированием.

В конечном счете принимаются решения о готовности программ, среды эксплуатационных площадок и пользователей к установке и эксплуатации первой рабочей версии системы.

Внедрение

На фазе внедрения программное обеспечение поставляется сообществу пользователей. Следует заметить, что они участвуют в демонстрации, обсуждении и испытании альфа- и бета-релизов проекта. Как только система попадает в руки конечного пользователя, часто возникают новые требования, предполагающие корректировку системы, решение ранее незамеченных проблем либо окончательную доработку функций, реализация которых была отложена. Обычно данная фаза начинается с выпуска бета-версии, впоследствии заменяемой рабочей версией системы.

Среди основных членов группы, выполняющих вышеизложенные задачи, – менеджер проекта, тестировщики, специалисты по выпуску версий, маркетологи и персонал, занятый в сфере продаж. Отметим, что работа по подготовке окончательной версии, маркетингу и продажам должна начинаться как можно раньше.

В конце фазы внедрения делается вывод о том, достигнуты ли цели проекта и стоит ли начинать следующий цикл разработки. Подводятся итоги работы и извлекаются уроки, которые помогут совершенствовать процесс разработки новых проектов.

Итерации

Каждая фаза RUP может быть разбита на итерации. *Итерация* представляет собой полный цикл разработки, по завершении которого выпускается промежуточная или итоговая версия работающего продукта, где реализована часть запланированных функций. От одной итерации к другой эта версия наращивается до получения готовой системы. Во время каждой итерации выполняются все дисциплины (см. ниже), хотя на разных фазах основной акцент делается на различных моментах. На начальной фазе главная задача заключается в сборе требований, на фазе разработки это анализ, проектирование и реализация архитектуры, на фазе конструирования – детальное проектирование, реализация и тестирование, а на фазе внедрения – размещение. Тестирование важно на всех стадиях.

Циклы разработки

Прохождение системы через четыре основные фазы называется *циклом разработки*. Каждый цикл завершается генерацией программного обеспечения. Первый проход через все фазы – это *начальный цикл разработки*. Если после него работа над проектом не прекращается, полученный продукт продолжает развиваться и снова минует те же фазы: начальный этап, разработку, конструирование и внедрение. Таким образом система эволюционирует, поэтому все циклы, следующие за начальным, называются *эволюционными*.

Дисциплины

RUP состоит из девяти дисциплин:

1. *Бизнес-моделирование* (business modeling) – описывает структуру и динамику организации-заказчика;
2. *Управление требованиями* (requirements) – выявляет требования на основе множества подходов;
3. *Анализ и проектирование* (analysis & design) – описывает множество архитектурных представлений системы;
4. *Реализация* (implementation) – собственно, разработка программного обеспечения, модульное тестирование и интеграция;
5. *Тестирование* (test) – описывает тестовые сценарии, процедуры и метрики для оценки дефектов;

6. *Размещение* (deployment) – включает спецификацию материалов, описание версии, обучение и другие аспекты, связанные с поставкой системы;
7. *Управление конфигурацией и изменениями* (configuration management) – сфокусировано на изменениях, поддержке целостности рабочих продуктов проекта и управленческой деятельности;
8. *Управление проектом* (project management) – описывает различные стратегии работы в условиях итерационного процесса;
9. *Создание рабочей среды* (environment) – рассматривает необходимую инфраструктуру для разработки системы.

Каждая дисциплина охватывает свой набор связанных рабочих продуктов и видов деятельности. *Рабочий продукт*[1] – это некоторый документ, отчет или исполнимая программа, которые после их создания преобразуются или потребляются. *Вид деятельности* описывает комплекс задач – этапы продумывания, выполнения, анализа, – выполняемых сотрудниками с целью создания или модификации рабочих продуктов, а также способы выполнения этих задач и соответствующие рекомендации. В число таких способов может входить применение инструментальных средств, позволяющих автоматизировать ряд процессов.

В некоторых из этих дисциплин уставновлены важные связи между рабочими продуктами. Например, модель вариантов использования, созданная в ходе сбора требований, *реализуется* в виде проектной модели, выработанной в ходе анализа и проектирования, *воплощается* в модели дисциплины реализации и *верифицируется* моделью дисциплины тестирования.

Рабочие продукты

С каждым видом деятельности в рамках RUP ассоциированы рабочие продукты (входные или выходные). Некоторые рабочие продукты используются как входные для последующих видов деятельности, содержат справочные сведения о проекте или придают ему окончательную форму для поставки по контракту.

Модели

Моделирование обсуждается в главе 1.

Модели – наиболее важная разновидность рабочих продуктов в RUP. Модель представляет собой упрощенное представление реальности, созданное для лучшего понимания разрабатываемой системы. В RUP существует множество моделей, которые в своей совокупности охватывают все важнейшие решения, направленные на визуализацию, специфицирование, конструирование и документирование программных систем:

1. *Модель бизнес-процессов* (business use case model) описывает деятельность организации;
2. *Модель бизнес-анализа* (business analysis model) определяет контекст системы;
3. *Модель вариантов использования* (use case model) выражает функциональные требования к системе;
4. *Модель анализа* (analysis model) – необязательная. Определяет проектные решения на концептуальном уровне;
5. *Проектная модель* (design model) охватывает словарь предметной области и области решения;
6. *Модель данных* (data model) – необязательная. Определяет представление данных в базах данных и других репозиториях;
7. *Модель размещения* (deployment model) специфицирует топологию аппаратных средств, на которых работает система, вместе с механизмами параллелизма и синхронизации;
8. *Модель реализации* (implementation model) определяет, какие части используются для сборки и реализации физической системы.

Архитектура обсуждается в главе 2.

Представление (view) – это проекция модели. В RUP архитектура системы включает пять тесно связанных друг с другом представлений: проектирования, взаимодействия, размещения, реализации и вариантов использования.

Другие рабочие продукты

Рабочие продукты RUP подразделяются на две группы: продукты управления и технические продукты. Последние, в свою очередь, делятся на пять основных подгрупп:

1. *Группа требований* (requirements set) описывает, что должна делать система. В составе этих рабочих продуктов могут быть модель вариантов использования, нефункциональные требования, модель предметной области, модель анализа и другие формы выражения потребностей пользователей, включая макеты, прототипы интерфейсов, юридические ограничения и т.д.;
2. *Группа анализа и проектирования* (analysis & design set) описывает, как система должна быть построена с учетом всех ограничений по времени, бюджету, унаследованным системам, повторному использованию, требованиям к качеству

[1] В отличие от языка UML, в контексте процесса разработки RUP термин «artifact» по смыслу целесообразнее переводить как «рабочий продукт», а не «артефакт». – *Прим. ред.*

и т.п. Сюда могут входить проектная модель, модель тестирования и другие формы выражения природы системы, включая прототипы и исполняемые архитектуры, но не ограничиваясь ими;

3. *Группа тестирования* (test set) описывает подход, посредством которого система верифицируется и аттестуется. С этой целью используются скрипты, сценарии тестирования, способы измерения дефектов и критерии приемки;
4. *Группа реализации* (implementation set) дает информацию об элементах программного обеспечения, включая исходный код на разных языках программирования, файлы конфигурации, файлы данных и т.д., а также о сборке системы на основе разработанных программных компонентов;
5. *Группа размещения* (deployment set) представляет все данные, необходимые для конфигурирования поставляемой системы, – в частности, включает всю информацию о способах пакетирования, поставки, установки и запуска программного обеспечения на целевой платформе заказчика.

Глоссарий

Object Constraint Language (OCL) – формальный язык для выражения ограничений без побочных эффектов

Unified Modeling Language (UML) – унифицированный язык моделирования, предназначенный для визуализации, специфицирования, конструирования и документирования артефактов программных систем

Абстрактный класс (abstract class) – класс, для которого нельзя создать прямые экземпляры объектов

Абстракция (abstraction) – важная характеристика сущности, отличающая ее от всех прочих видов сущностей. Абстракция определяет границы сущности с определенной точки зрения

Автомат (state machine) – поведение, которое специфицирует последовательность состояний объекта, через которые он проходит на протяжении своего жизненного цикла, реагируя на события (в числе прочего дает описание реакций на эти события)

Агрегат (aggregate) – класс, представляющий целое в агрегации

Агрегация (aggregation) – разновидность ассоциации, описывающая связь между агрегатом (целым) и компонентом (частью)

Активный класс (active class) – класс, экземпляры которого являются активными объектами

Активный объект (active object) – объект, который владеет процессом или потоком и может инициировать управляющее воздействие

Аргумент (argument) – конкретное значение, соответствующее параметру

Архитектура (architecture) – совокупность существенных решений относительно организации программной системы:

- выбор структурных элементов и интерфейсов, из которых она состоит, вместе с их поведением, описываемым в терминах кооперации этих элементов;
- объединение данных структурных и поведенческих элементов в крупные подсистемы;
- архитектурный стиль, которому подчинена организация элементов, интерфейсов, кооперации и их композиция.

К архитектуре программного обеспечения относятся не только структура и поведение, но также используемость, функциональность,

производительность, гибкость, повторное использование, понятность, экономические и технологические ограничения и компромиссы, а также эстетические аспекты

Асинхронное действие (asynchronous action) – запрос, при котором посылающий объект не приостанавливается для ожидания ответа

Ассоциация (association) – структурная связь, описывающая набор ссылок, где некая ссылка представляет собой соединение между объектами. Семантическая связь между несколькими классификаторами, в которой участвуют соединения между их экземплярами

Ассоциация-класс (association class) – элемент модели, имеющий свойства как ассоциации, так и класса. Может быть представлен как ассоциация со свойствами класса либо как класс со свойствами ассоциации

Атрибут (attribute) – именованное свойство классификатора, описывающее диапазон значений, которые могут принимать экземпляры свойства

Бинарная ассоциация (binary association) – ассоциация между двумя классами

Булево выражение (Boolean expression) – выражение, результатом вычисления которого является булево значение

Булев (Boolean) – перечислимый тип, принимающий значения «истина» (true) или «ложь» (false)

Вариант использования (use case) – описание набора последовательностей действий (включая вариации), выполняемых системой, которые приводят к значимому результату для некоего действующего лица

Версия (release) – относительно полный и согласованный набор рабочих продуктов, поставляемый внутреннему или внешнему пользователю

Взаимодействие (interaction) – поведение, описываемое набором сообщений, которыми обмениваются между собой объекты в определенном контексте для достижения заданной цели

Видимость (visibility) – указание на то, видимо ли имя и каким образом оно может быть использовано

Внедрение (transition) – четвертая фаза жизненного цикла разработки программного обеспечения, в течение которой оно передается пользователям

Временная метка (timing mark) – обозначение момента времени, в который происходит событие

Временное выражение (time expression) – выражение, результатом вычисления которого является абсолютное или относительное значение времени

Временное ограничение (timing constraint) – семантическое утверждение об абсолютном или относительном значении времени или временного интервала

Временное событие (time event) – событие, обозначающее истечение определенного периода времени с момента входа в текущее состояние

Временный объект (transient object) – объект, который существует только до тех пор, пока выполняется создавший его процесс или поток

Время (time) – значение, представляющее абсолютный или относительный момент

Выполнение (execution) – однократный прогон динамической модели

Выражение (expression) – строка, вычисляющая значение определенного типа

Выражение типа (type expression) – выражение, определяющее ссылку на один или несколько классификаторов

Действие (action) – выполнимое атомарное вычисление, в результате которого изменяется состояние системы или возвращается значение

Действительный параметр (actual parameter) – аргумент функции или процедуры

Действующее лицо (actor) – множество логически связанных ролей, выполняемых пользователями при взаимодействии с вариантами использования

Делегирование (delegation) – способность объекта послать ответ на сообщение другого объекта

Деятельность (activity) – поведение, выраженное множеством действий, связанных потоками управления и данных

Диаграмма (diagram) – изображение набора элементов – чаще всего в виде связного графа, состоящего из вершин (сущностей) и дуг (связей)

Диаграмма вариантов использования (use case diagram) – диаграмма, показывающая набор вариантов использования, действующих лиц и их связей. Относится к статическому представлению системы с точки зрения вариантов использования

Диаграмма взаимодействия (interaction diagram) – диаграмма, обеспечивающая динамическое представление системы и показывающая некое взаимодействие, где занято множество объектов и связей между ними, включая сообщения, которыми они могут обмениваться. Этот обобщенный термин применяется к нескольким видам диаграмм, фокусирующихся на взаимодействии объектов, в частности к диаграммам коммуникации и последовательности. Диаграммы

деятельности также имеют отношение к диаграммам взаимодействия, но семантически отличаются от них

Диаграмма деятельности (activity diagram) – диаграмма, показывающая поток управления и данных от одной деятельности к другой. Относится к динамическому представлению системы

Диаграмма классов (class diagram) – диаграмма, показывающая набор классов, интерфейсов и кооперации с их связями. Относится к статическому представлению системы с точки зрения проектирования, демонстрируя совокупность декларативных (статических) элементов

Диаграмма коммуникации (communication diagram) – диаграмма взаимодействия, подчеркивающая структурную организацию объектов, отправляющих и принимающих сообщения. Показывает взаимодействия, связанные с экземплярами и ссылками между ними

Диаграмма компонентов (component diagram) – диаграмма, показывающая организацию зависимостей в наборе компонентов. Относится к статическому представлению системы с точки зрения реализации

Диаграмма объектов (object diagram) – диаграмма, показывающая набор объектов и их связей в некоторый момент времени. Относится к статическому представлению системы с точки зрения проектирования или процессов

Диаграмма последовательности (sequence diagram) – диаграмма взаимодействия, выделяющая временной порядок сообщений

Диаграмма размещения (deployment diagram) – диаграмма, которая показывает конфигурацию обрабатывающих узлов и размещенные на них компоненты. Относится к статическому представлению системы с точки зрения размещения

Диаграмма состояний (state diagram) – диаграмма, показывающая автомат (машину состояний). Относится к динамическому представлению системы

Динамическая классификация (dynamic classification) – семантическая вариация обобщения, в котором объект может изменять свой тип или роль

Динамическое представление (dynamic view) – аспект системы, выделяющий ее поведение

Дополнение (adornment) – деталь спецификации элемента, добавленная к его базовой графической нотации

Зависимость (dependency) – семантическая связь между двумя сущностями, при которой изменение одной из них (независимой) может повлиять на семантику другой (зависимой)

Задача (task) – единственный путь выполнения некоторой программы, динамической модели или иного представления потока управления; процесс или поток

Защитное условие (guard condition) – Условие, которое должно быть выполнено для срабатывания перехода между состояниями

Значение (value) – Элемент области определения типа

Иерархия вложенности (containment hierarchy) – иерархия в пространстве имен, состоящая из элементов и связей агрегации между ними

Импорт (import) – применительно к пакетам – зависимость, показывающая тот из них, на классы которого можно ссылаться внутри данного пакета (включая и рекурсивно вложенные в него пакеты) без указания квалифицированного имени

Имя (name) – название сущности, связи или диаграммы; строка, используемая для идентификации элемента

Интерфейс (interface) – набор операций, используемый для спецификации сервиса класса или компонента

Использование (usage) – зависимость, при которой для правильного функционирования одного элемента (клиента) требуется присутствие другого (сервера)

Итерационный (iterative) – в контексте цикла разработки программного обеспечения – процесс управления потоком работающих версий

Итерация (iteration) – четко определенный перечень работ с фиксированным планом и критериями оценки, после проведения которых выпускается промежуточная или итоговая версия системы

Каркас (framework) – архитектурный образец, представляющий расширяемый шаблон для приложений в некоторой предметной области

Квалификатор (qualifier) – атрибут ассоциации, значение которого разбивает набор объектов, связанных с некоторым объектом посредством данной ассоциации, на непересекающиеся подмножества

Класс (class) – описание множества объектов, обладающих общими атрибутами, операциями, связями и семантикой

Классификатор (classifier) – механизм, описывающий структурные и поведенческие свойства. Классификаторы включают в себя классы, интерфейсы, типы данных, сигналы, компоненты, узлы, варианты использования и подсистемы

Клиент (client) – классификатор, запрашивающий сервис от другого классификатора

Комментарий (comment) – аннотация, присоединенная к элементу или набору элементов

Композит (composite) – класс, связанный композицией с одним или несколькими классами

Композиция (composition) – форма агрегации со строгим владением и совпадением времени жизни частей некоего целого. Части с нефиксированной множественностью могут быть созданы позже самого композита, но после создания живут и умирают вместе с ним. Такие части могут быть явно удалены перед уничтожением композита

Компонент (component) – физическая и замещаемая часть системы, обеспечивающая реализацию заданного набора интерфейсов

Конец ассоциации (association end) – конечная точка ассоциации, соединяющая ее с классификатором

Конец ссылки (link end) – экземпляр конца ассоциации

Конечный автомат (state machine) – см. Автомат

Конкретный класс (concrete class) – класс, для которого могут быть созданы прямые объекты (в противоположность абстрактному)

Конструирование (construction) – третья фаза жизненного цикла разработки программного обеспечения, в ходе которой исполняемый архитектурный прототип доводится до состояния, в котором может быть передан пользователям

Контейнер (container) – объект, предназначенный для хранения других объектов и предоставляющий операции для доступа или итерации по содержащимся в нем элементам

Контекст (context) – набор взаимосвязанных элементов, предназначенных для определенной цели, к примеру для специфицирования операции

Кооперация (collaboration) – объединение ролей и других элементов, работающих совместно для обеспечения общего поведения, которое представляет нечто большее, чем поведение суммы всех тех же составляющих. Кооперация определяет, как элемент наподобие варианта использования или операции реализуется посредством набора классификаторов и ассоциаций, играющих определенные роли и используемых определенным образом

Линия жизни объекта (object lifeline) – линия на диаграмме последовательности, отражающая существование объекта в течение некоторого периода времени

Местоположение (location) – размещение артефакта на узле

Метакласс (metaclass) – класс, экземплярами которого являются другие классы

Метод (method) – реализация операции

Механизм (mechanism) – образец проектирования, применимый к объединению классов

Механизм расширения (extensibility mechanism) – один из трех механизмов (стереотипы, помеченные значения и ограничения), которые позволяют расширять UML контролируемым образом

Множественная классификация (multiple classification) – семантическая разновидность обобщения, при котором объект может непосредственно принадлежать нескольким классам

Множественное наследование (multiple inheritance) – семантическая вариация обобщения, при котором потомок может иметь нескольких родителей

Множественность (multiplicity) – спецификация допустимой мощности множества

Модель (model) – упрощенное представление реальности, предназначенное для лучшего понимания создаваемой системы; семантически замкнутая абстракция системы

Мощность множества (cardinality) – количество элементов в множестве

Наследование (inheritance) – механизм, посредством которого некий специализированный элемент включает в себя структуру и поведение более общего

Наследование интерфейса (interface inheritance) – наследование интерфейса специализированного элемента. Не включает наследование реализации

Наследование реализации (implementation inheritance) – наследование реализации более общего элемента. Также включает наследование интерфейса

n-арная ассоциация (n-ary association) – ассоциация между несколькими классами

Начальная фаза (inception) – первая фаза цикла разработки программного обеспечения, когда исходная идея становится достаточно обоснованной, чтобы можно было принять решение о переходе к фазе разработки

Неортогональное подсостояние (nonorthogonal substate) – подсостояние, в котором система не может находиться, одновременно пребывая в других подсостояниях того же составного состояния

Неполнота (incomplete) – моделирование элемента с пропуском некоторых его частей

Несогласованность (inconsistent) – моделирование элемента, для которого не гарантируется целостность модели

Поток (thread) – облегченный поток управления, который может выполняться параллельно с другими потоками в одном и том же процессе

Область действия (scope) – контекст, в котором употребление некоторого имени является осмысленным

Обобщение (generalization) – связь специализации/обобщения, при которой объекты специализированного элемента (потомка) являются замещающими для объектов обобщающего элемента (родителя)

Образец (pattern) – общее решение типичной проблемы в заданном контексте

Обратное проектирование (reverse engineering) – Процесс трансформации кода в модель посредством отображения из определенного языка реализации

Объект (object) – конкретная материализация абстракции; сущность с хорошо определенными границами и идентичностью, в которой инкапсулированы состояние и поведение; экземпляр класса

Обязанность (responsibility) – контракт или обязательства типа либо класса

Ограничение (constraint) – расширение семантики элемента UML, позволяющее добавлять новые правила или модифицировать существующие

Одиночное наследование (single inheritance) – семантическая вариация обобщения, при котором потомок может иметь только одного родителя

Операция (operation) – реализация сервиса, который может быть запрошен у любого объекта класса с целью воздействия на поведение данного объекта

Ортогональное подсостояние (orthogonal substate) – подсостояние, в котором система может находиться, одновременно пребывая в других подсостояниях того же составного состояния.

Отправитель (sender) – объект, посылающий сообщение

Отправка (send) – передача экземпляра сообщения от объекта-отправителя объекту-получателю

Пакет (package) – контейнер общего назначения, служащий для организации элементов в группы

Параллелизм (concurrency) – одновременное выполнение нескольких работ в течение одного и того же промежутка времени. Параллелизм может быть обеспечен чередующимся либо действительно одновременным выполнением нескольких потоков

Параметр (parameter) – спецификация переменной, которая может быть изменена, передана или возвращена

Параметризованный элемент (parameterized element) – дескриптор элемента с одним или несколькими несвязанными параметрами

Переход (transition) – связь между двумя состояниями, показывающая, что объект, находящийся в первом состоянии, должен выполнить некоторые действия и перейти во второе, как только

наступит определенное событие и при этом будут выполнены заданные условия

Перечисление (enumeration) – список именованных значений, используемый в качестве диапазона значений определенного типа атрибутов

Плавательная дорожка (swimlane) – раздел на диаграмме деятельности, создаваемый с целью разграничения обязанностей

Поведение (behavior) – спецификация исполнимого вычисления

Поведенческое свойство (behavioral feature) – динамическое свойство элемента, например операция и др.

Подкласс (subclass) – в связи обобщения – потомок, представляющий собой специализацию другого класса (родителя)

Подсистема (subsystem) – группирование элементов, часть которых составляет спецификацию поведения, реализуемого другими содержащимися в нем элементами

Подсостояние (substate) – состояние, являющееся частью составного состояния

Помеченное значение (tagged value) – расширение свойств стереотипа UML, которое позволяет включать новую информацию в спецификацию элемента с данным стереотипом

Постоянный объект (persistent object) – объект, продолжающий существовать по завершении процесса или потока, создавшего его

Постусловие (postcondition) – ограничение, которое должно быть истинным по завершении выполнения операции

Потомок (child) – подкласс или другой специализированный элемент, над которым есть более общий (родитель)

Пошаговый (incremental) – в контексте жизненного цикла разработки программного обеспечения – процесс, обеспечивающий непрерывную интеграцию системной архитектуры для производства версий, в ряду которых каждая новая усовершенствована по сравнению с предыдущей

Предметная область (domain) – область знаний или деятельности с характерным понятийным аппаратом, которым владеют профессионалы, работающие в данной сфере

Представление (view) – проекция модели, рассматриваемая с определенной позиции, при которой сущности, не являющиеся релевантными для данной точки зрения, опущены

Представление взаимодействия (interaction view) – представление системной архитектуры, выделяющее объекты, потоки и процессы, которые формируют механизмы параллелизма и синхронизации; наборы деятельностей и потоки сообщений, управления

и данных между ними. Также касается производительности, масштабируемости и пропускной способности системы

Представление вариантов использования (use case view) – представление системной архитектуры, сфокусированное на вариантах использования и описывающее поведение системы с точки зрения ее конечных пользователей, аналитиков и тестировщиков

Представление проектирования (design view) – представление системной архитектуры, выделяющее классы, интерфейсы и кооперации, которые формируют словарь проблемной области и области решения. Касается функциональных требований к системе

Представление размещения (deployment view) – представление системной архитектуры, выделяющее узлы, которые формируют аппаратную топологию системы. Описывает распределение, поставку и установку частей, составляющих физическую систему

Представление реализации (implementation view) – представление системной архитектуры, которое подчеркивает артефакты, используемые для сборки и реализации физической системы. Описывает управление конфигурацией ее версий, состоящих из частично независимых артефактов, которые могут быть собраны различными способами для создания работающей системы

Предусловие (precondition) – ограничение, которое должно быть истинным перед вызовом операции

Прием (receive) – обработка экземпляра сообщения, переданного объектом-отправителем

Приемник (receiver) – объект, которому отправлено сообщение

Примитивный тип (primitive type) – базовый тип, например целое число или строка

Примечание (note) – графический символ для изображения ограничений или комментариев, присоединенный к элементу или совокупности элементов

Продукт (product) – конечные или промежуточные результаты процесса разработки, например модели, код, документация, рабочие планы

Проекция (projection) – отображение множества на его подмножество

Производный элемент (derived element) – элемент модели, который может быть вычислен на основе другого элемента, но тем не менее включен в модель для ясности или для удобства проектирования, хотя и не привносит новой семантики

Происшествие (occurrence) – экземпляр события, локализованный во времени и пространстве в определенном контексте. Может инициировать переход состояний в автомате

Пространство имен (namespace) – область действия, в которой могут быть определены и использованы имена. Каждое имя в ней идентифицирует уникальный элемент

Процесс (process) – ресурсоемкий поток управления, выполняемый параллельно с другими процессами

Прямое проектирование (forward engineering) – процесс трансформации модели в код посредством отображения на определенный язык программирования

Псевдосостояние (pseudostate) – вершина автомата, которая выглядит как состояние, но отличается от последнего своим поведением. Существуют три типа псевдосостояний: начальное, конечное и историческое

Рабочий продукт (artifact) – дискретный элемент информации, порождаемый или используемый процессом разработки программного обеспечения либо существующей системой

Разработка (elaboration) – вторая фаза жизненного цикла разработки программного обеспечения, в которой определяются концепция продукта и его архитектура

Реализация (implementation) – конкретное воплощение контракта, объявленного интерфейсом; определение того, как нечто конструируется или вычисляется

Реализация (realization) – семантическая связь между классификаторами, один из которых формулирует условия контракта, а другой обязуется его выполнять

Родитель (parent) – суперкласс или другой элемент, общий по отношению к потомку или потомкам (специализированным элементам)

Роль (role) – элемент структуры в определенном контексте

Свойство (feature) – сущностьнаподобие операции или атрибута, инкапсулированная внутри другой сущности, например интерфейса, класса или типа данных

Связывание (binding) – создание элемента по шаблону путем подстановки аргументов вместо параметров шаблона

Связь (relationship) – семантическое соединение элементов

Сервер (supplier) – тип, класс или компонент, предоставляющий услуги, которые могут быть востребованы другими элементами

Сигнал (signal) – спецификация асинхронного воздействия, передаваемого от одного экземпляра к другому

Сигнатура (signature) – имя и параметры операции

Синхронный вызов (synchronous call) – запрос, после передачи которого объект-отправитель ожидает результата

Система (system) – набор элементов, организованных для достижения конкретной цели и описываемый рядом моделей (возможно, с разных точек зрения). Зачастую декомпозируется на несколько подсистем

Сконцентрированный на архитектуре (architecture-centric) – в контексте жизненного цикла разработки программного обеспечения – процесс, сосредоточенный на ранней разработке и стабилизации архитектуры ПО, а впоследствии использующий системную архитектуру в качестве основного рабочего продукта для концептуализации, конструирования, управления и эволюции разрабатываемой системы

Событие (event) – спецификация значимого происшествия, локализованного в пространстве и времени. В контексте автомата появление события может вызвать переход состояния

Сообщение (message) – спецификация передачи информации между объектами в расчете на то, что за ним последует некоторая деятельность. Прием сообщения обычно трактуется как экземпляр события

Составное состояние (composite state) – состояние, включающее в себя либо параллельные, либо непересекающиеся подсостояния

Состояние (state) – ситуация в жизненном цикле объекта, во время которой он удовлетворяет некоторому условию, выполняет определенную деятельность или ожидает некоего события

Спецификация (specification) – текстовое описание синтаксиса и семантики конкретного строительного блока. Декларативное описание того, чем является и что делает некая сущность

Срабатывание (fire) – переход между состояниями

Ссылка (link) – семантическое соединение объектов, экземпляр ассоциации

Статическая классификация (static classification) – семантическая разновидность обобщения, в котором объект не может изменять свой тип или роль

Статическое представление (static view) –аспект системы, выделяющий ее структуру

Стереотип (stereotype) – расширение словаря UML, позволяющее создавать новые виды строительных блоков, производные от существующих, но специфичные для конкретной задачи

Строка (string) – последовательность текстовых символов

Структурное свойство (structural feature) – статическое свойство элемента

Суперкласс (superclass) – см. Родитель

Сценарий (scenario) – конкретная последовательность действий, иллюстрирующая поведение

Тип (type) – связь между элементом и его классификацией

Тип данных (datatype) – тип, значения которого никак не идентифицированы. К таковым относятся примитивные встроенные типы (например, числа и строки), а также перечислимые типы (например булевский)

Трассировка (trace) – зависимость, которая показывает историческую или процессную связь между двумя элементами, представляющими одно и то же понятие, без указания правил вывода одного элемента из другого

Требование (requirement) – желаемые функциональность, свойство или поведение системы

Узел (node) – физический элемент, существующий во время исполнения и представляющий вычислительный ресурс, который обычно обладает памятью, а зачастую и процессором

Управляемый вариантами использования (use case-driven) – в контексте цикла разработки программного обеспечения – процесс, в котором варианты использования служат основным рабочим продуктом для формулирования желаемого поведения системы, для верификации и аттестации системной архитектуры, для тестирования и обмена информацией между участниками проекта

Управляемый риском (risk-driven) – в контексте цикла разработки программного обеспечения – процесс, в котором при выпуске каждой новой версии основное внимание уделяется выявлению и устранению факторов, представляющих наибольший риск для успешного завершения проекта

Уровень абстракции (level of abstraction) – место в иерархии абстракций, нисходящей от верхних уровней (более абстрактных) до нижних (более конкретных)

Уточнение (refinement) – связь, которая представляет более полное описание того, что ранее уже было специфицировано на определенном уровне детализации

Фаза (phase) – промежуток времени между двумя важными опорными точками процесса разработки, в течение которого должны быть достигнуты заранее поставленные четкие цели, артефакты доведены до готовности и принято решение о том, можно ли переходить к следующей фазе

Фокус управления (focus of control) – символ на диаграмме последовательности, показывающий период времени, в течение которого объект выполняет действие – непосредственно или через подчиненную операцию

Формальный параметр (formal parameter) – некоторый параметр

Характеристика (property) – именованное значение, характеризующее элемент

Целостность (integrity) – правильность и согласованность взаимодействия различных сущностей

Шаблон (template) – параметризованный элемент

Экземпляр (instance) – конкретная материализация абстракции; сущность, к которой могут быть применены операции. Обладает состоянием, в котором запоминаются результаты операций

Экспорт (export) – в контексте пакетов – действие, направленное на обеспечение видимости элемента извне пространства имен, которое его включает

Элемент (element) – атомарная составляющая модели

Элизия (elision) – моделирование элемента, одна или несколько частей которого скрыты для упрощения представления

Предметный указатель

Г

Д

З

И

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ч

Ш

Э

Я

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **post@abook.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.abook.ru**.

Оптовые закупки: тел. **(095) 258-91-94, 258-91-95**; электронный адрес **abook@abook.ru**.

Гради Буч, Джеймс Рамбо, Ивар Якобсон

Язык UML. Руководство пользователя

Главный редактор *Мовчан Д.А.*
dm@dmkpress.ru
Переводчик *Мухин Н.*
Научный редактор *Вендров А.М.*
Литературный редактор *Готлиб О.В.*
Верстка, иллюстрации *Татаринов А.Ю.*
Дизайн обложки *Татаринов А.Ю.*

Подписано в печать 23.08.2006. Формат 70×100 $^{1}/_{16}$.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 40,. Тираж 3000 экз.

Издательство ДМК Пресс
Web-сайт издательства www.dmk-press.ru
Электронный адрес издательства books@dmk-press.ru

Академия АйТи, 117218, Москва, ул. Кржижановского, д. 21а
Департамент учебно-методической литературы «АйТи-Пресс»
Электронные адреса: www.academy.it.ru; infobook@it.ru; itpress@it.ru