Александр Леоненков

САМОУЧИТЕЛЬ

UML

2-е издание

Объектно-ориентированный анализ

Проектирование программных систем

Диаграммы логического и физического моделирования

Документирование проекта в нотации UML

UML в Rational Rose 2002

Язык объектных ограничений OCL

Освой UML самостоятельно

Александр Леоненков

САМОУЧИТЕЛЬ UML

2-е издание

Санкт-Петербург

«БХВ-Петербург»

2004

УДК 681.3.068+800.92UML
ББК 32.973.26-018.1
Л47

Леоненков А. В.

Л47 Самоучитель UML. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2004. — 432 с.: ил.

ISBN 5-94157-342-1

Цель книги — помочь менеджерам и руководителям проектов, руководителям информационных служб, бизнес-аналитикам, корпоративным программистам и ведущим разработчикам самостоятельно освоить базовые концепции и понятия наиболее перспективной и современной методологии разработки корпоративных информационных систем для последующего применения полученных знаний в ходе выполнения реальных проектов и совершенствования бизнес-процессов с использованием соответствующих CASE-средств. Рассматриваются основы современной технологии унифицированного анализа и проектирования программных систем на языке UML. Подробно излагаются базовые понятия языка UML, необходимые для построения объектно-ориентированных моделей корпоративных программных систем с использованием специальной графической нотации. Приводятся конкретные рекомендации по изображению канонических диаграмм UML и рассматриваются особенности разработки моделей с помощью CASE-средства IBM Rational Rose 2002. Описывается нотация OCL — языка объектных ограничений, что делает книгу уникальной среди аналогичных изданий.

Для программистов

УДК 681.3.068+800.92UML
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Андрей Смышляев*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.02.04.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 46,44.
Тираж 3 000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-342-1

© Леоненков А. В., 2004
© Оформление, издательство "БХВ-Петербург", 2004

Содержание

Предисловие **9**

Структура книги 10
Рекомендации по изучению языка UML 11
Благодарности 12

ЧАСТЬ I. ОСНОВЫ UML **15**

Глава 1. Введение **17**

1.1. Методология процедурно-ориентированного программирования 17
1.2. Методология объектно-ориентированного программирования 22
1.3. Методология объектно-ориентированного анализа и проектирования 30
1.4. Методология системного анализа и системного моделирования 34

Глава 2. Исторический обзор развития методологии объектно-ориентированного анализа и проектирования сложных систем **39**

2.1. Предыстория. Математические основы 39
2.2. Диаграммы структурного системного анализа 51
2.3. Основные этапы развития UML 63

Глава 3. Основные компоненты языка UML **70**

3.1. Назначение языка UML 72
3.2. Общая структура языка UML 76
3.3. Пакеты в языке UML 79
3.4. Основные пакеты метамодели языка UML 82
3.5. Специфика описания метамодели языка UML 92
3.6. Особенности изображения диаграмм языка UML 97

ЧАСТЬ II. ДИАГРАММЫ КОНЦЕПТУАЛЬНОГО, ЛОГИЧЕСКОГО И ФИЗИЧЕСКОГО МОДЕЛИРОВАНИЯ 103

Глава 4. Диаграмма вариантов использования (use case diagram) 105

4.1. Вариант использования 106
4.2. Актеры 108
4.3. Примечания 111
4.4. Отношения на диаграмме вариантов использования 112
4.5. Расширение языка UML для бизнес-моделирования 118
4.6. Текстовые сценарии вариантов использования 121
4.7. Пример построения диаграммы вариантов использования для системы управления банкоматом 123
4.8. Рекомендации по разработке диаграмм вариантов использования 127

Глава 5. Диаграмма классов (class diagram) 133

5.1. Класс 134
5.2. Отношения между классами 145
5.3. Расширение языка UML для построения моделей программного обеспечения и бизнес-систем 158
5.4. Шаблоны или параметризованные классы 162
5.5. Пример построения диаграммы классов системы управления банкоматом 164
5.6. Рекомендации по построению диаграмм классов 165

Глава 6. Диаграмма кооперации (collaboration diagram) 169

6.1. Кооперация 170
6.2. Объекты 175
6.3. Связи 180
6.4. Сообщения 182
6.5. Пример построения диаграммы кооперации системы управления банкоматом 188
6.6. Заключительные рекомендации по построению диаграмм кооперации 190

Глава 7. Диаграмма последовательности (sequence diagram) 193

7.1. Объекты 193
7.2. Сообщения на диаграмме последовательности 197
7.3. Пример построения диаграммы последовательности 203

7.4. Пример построения диаграммы последовательности системы управления банкоматом ... 206
7.5. Заключительные рекомендации по построению диаграмм последовательности ... 208

Глава 8. Диаграмма состояний (statechart diagram) ... 210

8.1. Конечные автоматы ... 212
8.2. Состояние ... 215
8.3. Переход ... 219
8.4. Составное состояние и подсостояние ... 225
8.5. Исторические состояния ... 228
8.6. Сложные переходы ... 230
8.7. Пример построения диаграммы состояний системы управления банкоматом ... 235
8.8. Заключительные рекомендации по построению диаграмм состояний ... 237

Глава 9. Диаграмма деятельности (activity diagram) ... 240

9.1. Состояние действия ... 241
9.2. Переходы ... 243
9.3. Дорожки ... 249
9.4. Объекты ... 251
9.5. Пример построения диаграммы деятельности системы управления банкоматом ... 255
9.6. Рекомендации по построению диаграмм деятельности ... 257

Глава 10. Диаграмма компонентов (component diagram) ... 260

10.1. Компоненты ... 262
10.2. Интерфейсы ... 265
10.3. Зависимости ... 266
10.4. Пример построения диаграммы компонентов системы управления банкоматом ... 270
10.5. Рекомендации по построению диаграммы компонентов ... 272

Глава 11. Диаграмма развертывания (deployment diagram) ... 275

11.1. Узел ... 276
11.2. Соединения и зависимости ... 280
11.3. Пример построения диаграммы развертывания системы управления банкоматом ... 282
11.4. Рекомендации по построению диаграммы развертывания ... 284

ЧАСТЬ III. АНАЛИЗ И ПРОЕКТИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ НОТАЦИИ ЯЗЫКА UML И CASE-СРЕДСТВА IBM RATIONAL ROSE 2002 289

Глава 12. Общая характеристика инструментария IBM Rational Rose 2002 291

12.1. Общая характеристика CASE-средства IBM Rational Rose 2002 292
12.2. Особенности рабочего интерфейса IBM Rational Rose 2002 293
12.3. Назначение операций главного меню 299
12.4. Назначение операций стандартной панели инструментов 312

Глава 13. Начало работы над проектом в среде IBM Rational Rose 2002 315

13.1. Разработка диаграммы вариантов использования в среде IBM Rational Rose 315
13.2. Разработка диаграммы классов в среде IBM Rational Rose 325
13.3. Разработка диаграммы кооперации в среде IBM Rational Rose 333
13.4. Разработка диаграммы последовательности в среде IBM Rational Rose 340

Глава 14. Завершение работы над проектом в среде IBM Rational Rose 2002 346

14.1. Разработка диаграммы состояний в среде IBM Rational Rose 346
14.2. Разработка диаграммы деятельности в среде IBM Rational Rose 352
14.3. Разработка диаграммы компонентов в среде IBM Rational Rose 357
14.4. Разработка диаграммы развертывания в среде IBM Rational Rose 362
14.5. Генерации программного кода в среде IBM Rational Rose 366

Заключение 375

Приложение 379

Язык объектных ограничений 379
Выражения языка OCL 381
Основные типы значений и операций в языке OCL 383

Операции над отдельными типами значений 384
Допустимые выражения в языке OCL 391
Неопределенные значения выражений 392
Совокупности допустимых значений в языке OCL 392
Операции над совокупностями значений 393
Некоторые операции с множествами, последовательностями и комплектами 396
Операции преобразования типов 397
Примеры записи выражений языка OCL 398

Глоссарий 401

Литература 417

Предметный указатель 421

Предисловие

Книга посвящена рассмотрению основ Унифицированного языка моделирования или, сокращенно, языка UML (Unified Modeling Language), который предназначен для описания, визуализации и документирования объектно-ориентированных систем и бизнес-процессов с ориентацией на их последующую реализацию в виде программного обеспечения.

Чем обусловлена необходимость создания и изучения еще одного языка, который вовсе не является языком программирования в привычном смысле этого слова?

Прежде всего — это возрастающая сложность прикладных программ и высокая стоимость их разработки и сопровождения. Действительно, в разработке современных корпоративных информационных систем принимают участие десятки и сотни различных специалистов, для которых построение предварительной модели системы до начала написания соответствующего программного кода стало настоятельной необходимостью. Разработка программных систем на заказ также приводит к необходимости поддержания единого стиля для различных версий программ при их постоянной доработке и модификации.

Основное требование к такой предварительной модели программной системы — модель должна быть понятна как заказчику, так и всем специалистам проектной группы, включая и программистов. Оказалось, что разработка соответствующего языка моделирования или нотации является непростым делом. Потребовалось несколько лет, прежде чем усилия группы специалистов ведущих фирм-производителей программного и аппаратного обеспечения привели к появлению языка UML.

Какими бы свойствами ни обладал новый язык, важной особенностью для его жизнеспособности является реализация и поддержка соответствующей нотации в коммерческих программных продуктах. В последние годы наблюдается активный интерес к языку UML, о чем свидетельствуют десятки

коммерческих программных инструментариев, предназначенных для автоматизации разработки программного обеспечения на основе построения предварительной объектно-ориентированной модели предметной области. Одним из наиболее мощных инструментальных средств этого класса по-прежнему остается "широко известный в узких корпоративных кругах" IBM Rational Rose.

Хотя в настоящее время в России получили широкое применение три нотации визуального моделирования: IDEF (Icam DEFinition), ARIS (Architecture of Integrated Information Systems) и UML, именно последняя из них все более активно используется корпоративными программистами для разработки графических моделей при выполнении программных проектов. При этом новые инструментарии разработки приложений и соответствующие среды визуального программирования MS Visual Studio 6/.NET и Borland Delphi/C++Builder/Jbuilder не только поддерживают нотацию языка UML в качестве базового средства моделирования программных систем, но и позволяют получить исполнимые модули программ на основе разработанной графической модели.

Следует отметить еще одну область, в которой все более активно используются графические нотации — это выполнение работ по приведению системы менеджмента качества в соответствии со стандартом ISO 9001:2000 в рамках сертификации предприятий и компаний. В этой области язык UML применяется для визуального моделирования и документирования бизнес-процессов, в результате чего разработанные диаграммы и пояснения к ним предоставляются международным сертификационным органам для получения соответствующего сертификата.

Структура книги

Во втором издании книги были исправлены некоторые неточности в тексте и на графических диаграммах, уточнена семантика отдельных терминов и даны более ясные формулировки основных элементов языка UML. При этом материал книги соответствует последней версии языка UML 1.5. Для иллюстрации рассматриваемых понятий языка UML приводится описание сквозного примера модели системы управления банкоматом.

В основу книги положены две основные идеи. С одной стороны, рассмотреть все базовые конструкции языка UML, необходимые для самостоятельной разработки концептуальных, логических и физических моделей программного обеспечения. С другой стороны, донести до читателя основы методологии моделирования сложных систем, без понимания которой вряд ли возможно адекватно и безошибочно использовать богатейший потенциал возможностей языка UML.

Материал книги делится на три части. Первая часть знакомит с основными теоретическими понятиями, которые необходимы для правильного понимания назначения и возможностей языка UML. Здесь также приводится исторический обзор развития технологий программирования и методологии объектно-ориентированного анализа и проектирования. Поскольку UML не является формальным языком с фиксированным синтаксисом, описание языка рассматривается как некоторая открытая модель с определенными базовыми семантическими конструкциями и неформальными правилами их расширения.

Вторая часть является центральной в книге и содержит описание назначения элементов всех канонических диаграмм языка UML (версия 1.5), которые являются основой построения концептуальных, логических и физических моделей. В отдельных главах второй части последовательно рассматриваются:

- диаграмма вариантов использования;
- диаграмма классов;
- диаграмма кооперации;
- диаграмма последовательности;
- диаграмма состояний;
- диаграмма деятельности;
- диаграмма компонентов;
- диаграмма развертывания.

Для каждой из диаграмм описываются базовые элементы графической нотации, необходимые для изображения различных элементов диаграмм, а также рассматривается соответствующая диаграмма сквозного примера модели системы управления банкоматом с использованием соответствующей графической нотации.

Третья часть содержит описание особенностей реализации языка UML в уже упомянутом CASE-инструментарии — IBM Rational Rose 2002. В заключении сделана попытка оценить перспективы дальнейшего развития языка UML и технологии компонентной разработки приложений.

Рекомендации по изучению языка UML

Тематика книги, возможно, производит нетрадиционное впечатление для многих программистов, занятых разработкой конкретных приложений. Однако современные тенденции развития индустрии создания программного обеспечения складываются таким образом, что именно язык UML де-факто оказывается общепризнанным стандартом в области разработки моделей

систем и процессов с его последующей реализацией в соответствующих инструментальных средствах.

В то же время, следует отметить одну важную особенность современной программной инженерии — это тенденция специализации в области разработки программ. Речь идет о появлении таких специалистов, как системный аналитик, менеджер проекта, бизнес-аналитик и архитектор системы, которые, наряду со знанием языков программирования, должны владеть методологией объектно-ориентированного анализа и моделирования предметной области. Для системного инженера и интегратора также важно разбираться в возможностях реализации конкретных проектов и использовать общепринятую систему обозначений для решения своих задач. Наконец, программисты, занятые в масштабных проектах, должны четко понимать функциональные аспекты будущей программной системы. Для всех этих категорий специалистов и предназначена данная книга, поскольку язык UML позволяет организовать эффективное и понятное для всех общение.

Вполне вероятно, что со временем элементы нотации языка UML будут использоваться в учебных программах для обучения студентов самых различных специальностей. Во всяком случае, многие преподаватели давно осознали ограниченность существующих отечественных стандартов на разработку программной документации, которые не отражают современных тенденций развития программной инженерии. Как студенты, так и преподаватели найдут в книге интересный материал для размышления, который позволит понять целый ряд особенностей и перспектив профессионального образования в области современных информационных технологий.

Для понимания основных конструкций языка UML достаточно общей эрудиции и некоторого знакомства с одним из языков программирования. Читатели, которые ставят перед собой такую цель, могут бегло просмотреть материал первой части и сразу приступить к изучению отдельных диаграмм. Однако для творческого овладения методологией объектно-ориентированного анализа и моделирования необходима, как представляется, определенная математическая культура и знание некоторых общих понятий прикладного системного анализа. Соответствующий теоретический материал приводится в первой части книги и далее используется по мере изложения элементов конкретных диаграмм.

Благодарности

Автор искренне благодарит К. Н. Ильинского, Е. В. Кондукову, Е. В. Строганову, А. М. Коновалова и И. А. Корнеева за предоставленные в разное время материалы, которые были использованы при написании книги. Автор искренне признателен директору Школы IT-менеджмента АНХ при Правительстве РФ (**www.itmane.ru**) А. И. Соколову, а также Л. А. Ермакову, В. А. Перекрестову

и В. В. Фамильнову за оказанную поддержку в процессе работы над книгой. В предоставлении демонстрационной версии IBM Rational Rose 2002 неоценимую помощь оказали сотрудники компании Интерфейс Ltd., которым автор выражает свою признательность.

Написание современной книги немыслимо без использования ресурсов Интернета. В этой связи хотелось бы выразить особую признательность директору Междисциплинарного Центра СПбГУ (**www.icape.nw.ru**) профессору Н. В. Борисову за предоставленную возможность электронной коммуникации.

В заключение следует специально отметить одно немаловажное обстоятельство, которое усложняет понимание и распространение идей визуального моделирования среди отечественных системных аналитиков, менеджеров проектов, системных инженеров и программистов. Речь идет о неустановившейся терминологии в этой области и о неоднозначности перевода отдельных терминов, имеющих зачастую многозначное толкование в том или ином конкретном контексте. С этой целью названия наиболее важных понятий и их краткая характеристика отдельно приводятся в *Глоссарии* в конце книги. В любом случае автор будет признателен за все отзывы и конструктивные предложения, связанные с содержанием книги и проблематикой моделирования в контексте языка UML, которые можно отправлять по адресу: **uml@itmane.ru**.

Часть I

Основы UML

Глава 1. Введение

Глава 2. Исторический обзор развития методологии объектно-ориентированного анализа и проектирования сложных систем

Глава 3. Основные компоненты языка UML

Глава 1

Введение

Мир компьютерных и информационных технологий без преувеличения можно назвать наиболее динамичной областью современных знаний. Практически каждый год появляются новые модели процессоров и комплектующих, новые версии операционных систем и программного обеспечения. Все это происходит на фоне постоянного усложнения не только отдельных физических и программных компонентов, но и лежащих в их основе концепций и идей.

Кажется, еще совсем недавно профессиональному программисту было достаточно в совершенстве владеть одним-двумя языками программирования, чтобы разрабатывать серьезные программные приложения. Выбор платформы и операционной системы, как правило, не являлся серьезной проблемой. Разработка блок-схем алгоритмов и программ представлялась лишней тратой времени, и в лучшем случае выполнялась с целью документирования проекта. А сопровождение программы, хотя и было сопряжено с объективными трудностями, могло быть реализовано простым добавлением или изменением исходного кода.

1.1. Методология процедурно-ориентированного программирования

Появление первых электронных вычислительных машин или компьютеров ознаменовало новый этап в развитии техники вычислений. Казалось, достаточно разработать последовательность элементарных действий, каждое из которых преобразовать в понятные компьютеру инструкции, и любая вычислительная задача может быть решена. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала над всем процессом разработки программ. Появились специальные языки программирования, позволившие преобразовывать отдельные вычислительные операции в соответствующий программный код.

Основой данной методологии разработки программ являлась *процедурная* или *алгоритмическая организация* структуры программного кода. Это было

настолько естественно для решения вычислительных задач, что ни у кого не вызывала сомнений целесообразность такого подхода. Исходным понятием этой методологии являлось понятие *алгоритма*, под которым, в общем случае, понимается некоторое предписание выполнить точно определенную последовательность действий, направленных на достижение заданной цели или решение поставленной задачи. Примерами алгоритмов являются хорошо известные правила нахождения корней квадратного уравнения или корней линейной системы уравнений.

Примечание

Принято считать, что сам термин алгоритм, а точнее его более ранний вариант — *алгорифм*, происходит от имени средневекового математика Аль-Хорезми, который в 825 г. описал правила выполнения арифметических действий в десятичной системе счисления.

С этой точки зрения вся история математики тесно связана с разработкой тех или иных алгоритмов решения актуальных для своей эпохи задач. Более того, само понятие алгоритма стало предметом соответствующей теории — *теории алгоритмов*, которая занимается изучением общих свойств алгоритмов. Со временем содержание этой теории стало настолько абстрактным, что понимание соответствующих результатов стало доступно только специалистам. Как дань этой традиции какой-то период языки программирования назывались алгоритмическими, а первое графическое средство документирования программ получило название *блок-схемы алгоритма*. Соответствующая система графических обозначений была зафиксирована в ГОСТ 19.701-90, который регламентировал использование условных обозначений в схемах алгоритмов, программ, данных и систем.

Однако потребности практики не всегда требовали установления вычислимости конкретных функций или разрешимости отдельных задач. В языках программирования возникло и закрепилось новое понятие *процедуры*, которое конкретизировало общее понятие алгоритма применительно к решению задач на компьютерах. Так же, как и алгоритм, процедура представляет собой законченную последовательность действий или операций, направленных на решение отдельной задачи. В языках программирования появилась специальная синтаксическая конструкция, которая получила название процедуры.

Со временем разработка больших программ превратилась в серьезную проблему и потребовала их разбиения на более мелкие фрагменты. Основой для такого разбиения стала *процедурная декомпозиция*, при которой отдельные части программы или *модули* представляли собой совокупность процедур для решения некоторой совокупности задач. Главная особенность процедурного программирования заключается в том, что программа всегда имеет начало во времени или начальную процедуру (начальный блок) и окончание

(конечный блок). При этом вся программа может быть представлена визуально в виде направленной последовательности графических примитивов или блоков (рис. 1.1).

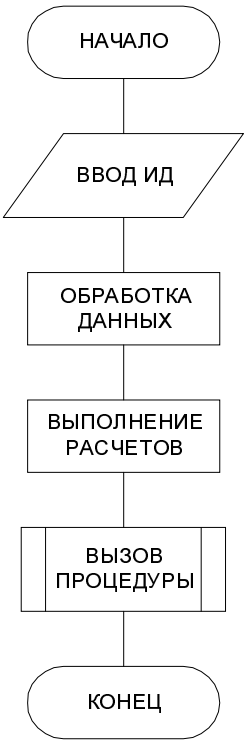


Рис. 1.1. Графическое представление программы в виде последовательности процедур

Важным свойством таких программ является необходимость завершения всех действий предшествующей процедуры для начала действий последующей процедуры. Изменение порядка выполнения таких действий даже в пределах одной процедуры потребовало включения в языки программирования специальных условных операторов типа `If-then-else` и `Goto` для реализации ветвления вычислительного процесса в зависимости от промежуточных результатов решения задачи.

Примечание

Появление и интенсивное использование условных операторов и оператора безусловного перехода стало предметом острых дискуссий среди специалистов по программированию. Дело в том, что бесконтрольное применение в программе оператора безусловного перехода `goto` способно серьезно осложнить

понимание кода. Соответствующие программы стали сравнивать со спагетти, называя их *bowl of spagetti*, имея в виду многочисленные переходы от одного фрагмента программы к другому или, что еще хуже, возврат от конечных операторов программы к ее начальным операторам. Ситуация казалась настолько драматичной, что в литературе зазвучали призывы исключить оператор `goto` из языков программирования. Именно с этого времени принято считать хорошим стилем программирование без `goto`.

Рассмотренные идеи способствовали становлению определенной системы взглядов на процесс разработки программ и написания программных кодов, которая получила название *методологии структурного программирования*. Основой данной методологии является процедурная декомпозиция программной системы и организация отдельных модулей в виде совокупности выполняемых процедур. В рамках данной методологии получило развитие *нисходящее проектирование* программ или программирование "сверху-вниз". Период наибольшей популярности идей структурного программирования приходится на конец 70-х — начало 80-х годов прошлого столетия.

Как вспомогательное средство структуризации программного кода было рекомендовано использование отступов в начале каждой строки, которые должны выделять вложенные циклы и условные операторы. Все это призвано способствовать пониманию или читабельности самой программы. Данное правило со временем было реализовано в большинстве популярных средств разработки. Листинг 1.1 фрагмента программы на языке Pascal иллюстрирует эту особенность написания программ.

Листинг 1.1. Пример фрагмента программы на языке Pascal, разработанной с использованием правил структурного программирования

```
Procedure FirstOpt;
  Begin
    FuncRaz(Frec, Rn);
    for i:=1 to N do
       RvarRec[i]:= Rn[i];
    FvarRec:= Frec;
    NumIt:=0;
      Repeat
        NumIt:=NumIt+1;
        V:= Frec;
        for j:=1 to K do
           for l:=1 to M do
              begin
                 S:=0.0;
```

```
                T:=0.0;
                for i:=1 to N do
                   begin
                      T:=T+sqr(W1[i,j])*Xpr[i,l];
                      S:=S+sqr(W1[i,j])
                   end;
                Zentr[j,l]:=T/S
             end;
          for j:=1 to K do
             for i:=1 to N do
                begin
                   S:=0.0;
                   P:=0.0;
                   Q:=0.0;
                   for l:=1 to M do
                      S:=S+sqr(Xpr[i,l]-Zentr[j,l]);
                    P:=1.0/S;
                end;
          Q:=0.0;
          D:=0;
          for i:=1 to N do
             for j:=1 to K do
                if Abs(W1[i,j]-W2[i,j]) >= Eps then D:=1;
          for i:=1 to N do
             for j:=1 to K do
                W1[i,j]:=W2[i,j]
      Until (D=0)or(NumIt=NumMax)
   End;
```

В этот период основным показателем сложности разработки программ считали ее размер. Вполне серьезно обсуждались такие оценки сложности программ, как количество строк программного кода. Правда, при этом делались некоторые предположения относительно синтаксиса самих строк, которые должны были удовлетворять определенным правилам. Общая трудоемкость разработки программ оценивалась специальной единицей измерения — "человеко-месяц" или "человеко-год". А профессионализм программиста напрямую связывался с количеством строк программного кода, который он мог написать и отладить в течение, скажем, месяца.

Примечание

Сейчас попытки оценить профессионализм программиста количеством строк программного кода могут вызвать лишь улыбку собеседника. Действительно, пользуясь встроенными мастерами современных инструментариев разработки (MS Visual C++ или Borland Delphi), даже новичок может за считанные секунды последовательным нажатием кнопок диалоговых меню создать работоспособное приложение, содержащее сотни строк программного кода и состоящее из десятка отдельных файлов проекта.

1.2. Методология объектно-ориентированного программирования

Со временем ситуация стала существенно изменяться. Оказалось, что трудоемкость разработки программных приложений на начальных этапах программирования оценивалась значительно ниже реально затрачиваемых усилий, что служило причиной дополнительных расходов и затягивания окончательных сроков готовности программ. В процессе разработки приложений изменялись функциональные требования заказчика, что еще более отдаляло момент окончания работы программистов. Увеличение размеров программ приводило к необходимости привлечения дополнительных программистов, что, в свою очередь, потребовало дополнительных ресурсов для организации их согласованной работы.

Но не менее важными оказались качественные изменения, связанные со смещением акцента использования компьютеров. Если в эпоху "больших машин" основными потребителями программного обеспечения были крупные предприятия, компании и учреждения, то позже появились персональные компьютеры и быстро стали повсеместным атрибутом мелкого и среднего бизнеса. Вычислительные и расчетно-алгоритмические задачи в этой области традиционно занимали второстепенное место, а на первый план выступили задачи обработки и манипулирования данными.

Стало очевидным, что традиционные методы процедурного программирования не способны справиться ни с растущей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов прошлого столетия возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Такой методологией стало *объектно-ориентированное программирование* (ООП).

Фундаментальными понятиями ООП являются понятия класса и объекта. При этом под *классом* понимают некоторую абстракцию совокупности объектов, которые имеют общий набор свойств и обладают одинаковым поведением. Каждый *объект* в этом случае рассматривается как экземпляр соответствующего класса. Объекты, которые не имеют полностью одинаковых

свойств или не обладают одинаковым поведением, по определению не могут быть отнесены к одному классу.

Примечание

Приведенное выше определение класса является общим. В последующих главах по мере изучения материала этот термин будет уточняться на основе установления семантических связей с другими понятиями объектно-ориентированного анализа и проектирования.

Важной особенностью классов является возможность их организации в виде некоторой *иерархической* структуры, которая по внешнему виду напоминает схему классификации понятий формальной логики. В этой связи следует заметить, что каждое понятие в логике имеет некоторый объем и содержание. При этом под *объемом* понятия понимают все другие мыслимые понятия, для которых исходное понятие может служить определяющей категорией или главной частью. *Содержание* понятия составляет совокупность всех его признаков или атрибутов, отличающих данное понятие от всех других. В формальной логике имеет место закон обратного отношения: если содержание понятия *A* содержится в содержании понятия *B*, то объем понятия *B* содержится в объеме понятия *A*.

Иерархия понятий строится следующим образом. В качестве наиболее общего понятия или категории берется понятие, имеющее наибольший объем и, соответственно, наименьшее содержание — это самый высокий уровень абстракции для данной иерархии. Затем данное общее понятие некоторым образом конкретизируется, тем самым уменьшается его объем и увеличивается содержание. Появляется менее общее понятие, которое на схеме иерархии будет расположено на уровень ниже исходного понятия. Этот процесс конкретизации понятий может быть продолжен до тех пор, пока на самом нижнем уровне не будет получено понятие, дальнейшая конкретизация которого в данном контексте либо невозможна, либо нецелесообразна.

Примерами наиболее общих понятий могут служить такие абстрактные категории, как система, структура, интеллект, информация, сущность, связь, состояние, событие и многие другие. В процессе изучения этих категорий появляются новые особенности их содержания и объема. Именно по этим причинам всегда трудно дать им точное определение. В качестве примеров конкретных понятий можно привести понятие книги, которую читатель держит в руках, или понятие микропроцессора Intel Pentium IV.

Примечание

Как будет видно из дальнейшего изложения, иерархическая схема организации понятий во многом похожа на рассматриваемую далее иерархию классов, но не тождественна ей. В общем случае взаимосвязи между классами могут иметь и другие качественные особенности. С другой стороны, иерархия понятий является более общей категорией по сравнению с иерархией уровней абстракции классов ООП.

Основными принципами ООП являются наследование, инкапсуляция и полиморфизм. Первый принцип, в соответствии с которым знание о более общей категории разрешается применять для более частной категории, называется *наследованием*. Наследование тесно связано с иерархией классов, которая определяет, какие классы следует считать наиболее абстрактными и общими по отношению к другим классам. При этом, если некоторый общий или родительский класс (предок) обладает фиксированным набором свойств и поведением, то производный от него класс (потомок) должен содержать этот же набор свойств и поведение, а также дополнительные, которые будут характеризовать уникальность полученного таким образом класса. В этом случае говорят, что производный класс наследует свойства и поведение родительского класса.

Для иллюстрации принципа наследования можно привести следующий пример. Рассмотрим в качестве общего класс "Автомобиль". Этот класс определяется как некоторая абстракция свойств и поведения всех реально существующих автомобилей. При этом свойствами класса "Автомобиль" могут быть такие общие свойства, как наличие двигателя, трансмиссии, колес и рулевого управления. Если в качестве производного класса рассмотреть класс "Легковой автомобиль", то все выделенные выше свойства будут присущи и этому классу. Можно сказать, что класс "Легковой автомобиль" наследует свойства родительского класса "Автомобиль". Однако кроме перечисленных свойств класс-потомок будет содержать дополнительные свойства, например, такое как наличие салона с количеством посадочных мест 2—5.

В свою очередь, класс "Легковой автомобиль" может быть классом-предком для других классов, которые вполне могут соответствовать, например, моделям конкретных фирм-производителей. С этой точки зрения можно рассматривать класс "Легковой автомобиль производства ВАЗ". Поскольку Волжский автомобильный завод выпускает несколько моделей автомобилей, одним из производных классов для предыдущего класса может быть конкретная модель автомобиля, например, ВАЗ-2110.

В этом контексте конкретный автомобиль, изготовленный на Волжском автомобильный заводе, имеет свои особенности, включая уникальный заводской номер, который отличает один автомобиль от другого. В этом случае автомобиль с идентификационным номером, например, XTA-211000V0001294, будет представлять собой один из объектов или экземпляров класса "Модель ВАЗ-2110".

Описанная выше информация о соотношении классов в нашем примере обладает одним серьезным недостатком, а именно, отсутствием наглядности. В этой связи возникает вопрос: а возможно ли представить иерархию наследования классов в визуальной форме? Традиционно для изображения понятий в формальной логике использовались окружности или прямоугольники. Используя эту графическую нотацию, иерархия классов для рассмотренного

примера может быть представлена в виде вложенных прямоугольников, каждый из которых соответствует отдельному классу (рис. 1.2).

Рис. 1.2. Иерархия вложенности классов для примера "Автомобиль"

Появление объектно-ориентированных языков программирования было связано с необходимостью реализации концепции классов и объектов на синтаксическом уровне. С точки зрения ООП, класс является дальнейшим расширением *структуры* (structure) или *записи* (record). Включение в известные языки программирования C и Pascal классов и некоторых других возможностей привело к появлению, соответственно, C++ и Object Pascal, которые, на сегодня, являются наиболее распространенными языками разработки приложений. Распространению C++ и Object Pascal способствовало то обстоятельство, что язык C++ был выбран в качестве базового для программного инструментария MS Visual C++, а язык Object Pascal — для популярного средства быстрой разработки приложений Borland Delphi.

За короткий период оба инструментария превратились в мощные системы разработки программ с соответствующими библиотеками стандартных классов, содержащих сотни различных свойств и методов. Применительно к среде MS Visual C++ 6/.NET такая библиотека имеет специальное название — MFC (Microsoft Foundation Classes), т. е. фундаментальные классы

от Microsoft. При этом производные классы наследуют свойства и методы родительских классов. Ниже приводится фрагмент иерархии классов MFC в том виде, как он изображен в соответствующей документации (рис. 1.3).

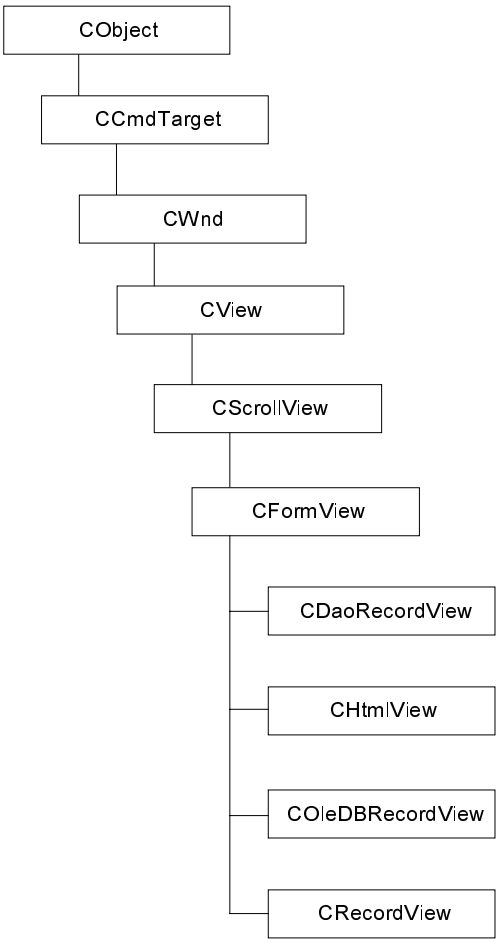


Рис. 1.3. Фрагмент иерархии классов MFC, используемых в среде программирования MS Visual C++ 6/.NET

Процесс разработки программ в среде Borland Delphi также тесно связан с использованием библиотеки стандартных классов — VCL (Visual Component Library) или библиотеки визуальных компонентов. Эта библиотека тоже построена по иерархическому принципу, в соответствии с которым компоненты нижележащих уровней наследуют свойства и методы вышележащих компонентов. Для этого случая также приводится фрагмент иерархии классов VCL (рис. 1.4).

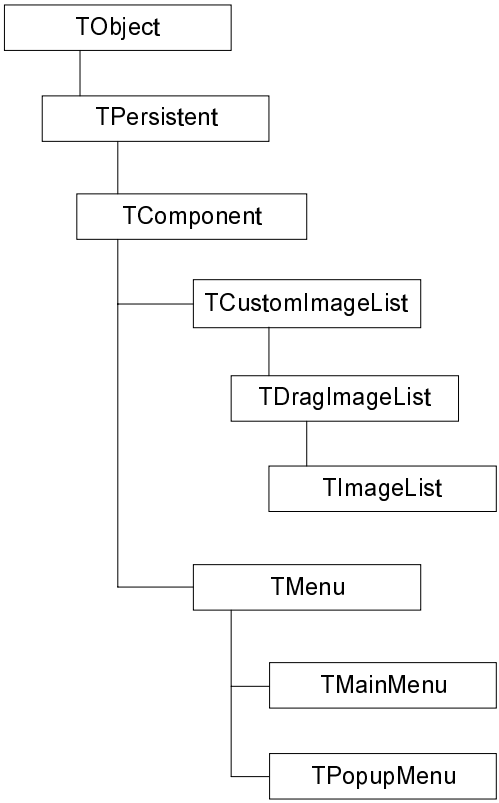


Рис. 1.4. Фрагмент иерархии классов VCL, используемых в среде программирования Borland Delphi 6/7

Даже этих простых примеров достаточно, чтобы понять следующий факт. А именно то, что для одной и той же общей концепции иерархии классов используются совершенно различные графические средства. В первом случае — вложенные прямоугольники, во втором — связные прямоугольники. В действительности, различных способов изображения классов предложено гораздо больше, небольшая часть которых будет рассмотрена далее. Однако уже сейчас важно осознать, что подобную ситуацию следовало бы унифицировать, т. е. использовать для этой цели некоторую единую систему обозначений.

Следующий принцип ООП — *инкапсуляция*. Этот термин характеризует сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей. Действительно, взаимодействующему с классом клиенту нет необходимости знать, каким образом реализован тот или иной метод класса, услугами которого он решил воспользоваться. Конкретная реализация присущих классу свойств и методов, которые определяют поведение этого класса, является собственным делом данного класса. Более того, отдельные свойства и методы класса вообще

могут быть невидимы за пределами этого класса, что является базовой идеей введения различных категорий видимости для компонентов класса.

Если продолжить рассмотрение примера с классом "Легковой автомобиль", то нетрудно проиллюстрировать инкапсуляцию следующим образом. Основным субъектом, который взаимодействует с объектами этого класса, является водитель. Вполне очевидно, что не каждый водитель в совершенстве знает внутреннее устройство легкового автомобиля. Более того, отдельные детали этого устройства сознательно скрыты в корпусе двигателя или коробке передач. А в случае нарушения работы автомобиля, являющейся причиной неадекватности его поведения, необходимый ремонт выполняет профессиональный механик.

Инкапсуляция ведет свое происхождение от деления модулей в некоторых языках программирования на две части (или секции): интерфейс и реализацию. При этом в интерфейсной секции модуля описываются все объявления функций и процедур, а возможно и типов данных, доступных за пределами данного модуля. Другими словами, указанные процедуры и функции являются способами оказания услуг внешним клиентам. В другой секции модуля, называемой реализацией, содержится программный код, который определяет конкретные способы реализации объявленных в интерфейсной части процедур и функций.

Принцип разделения модуля на интерфейс и реализацию отражает суть наших представлений об окружающем мире. В интерфейсной части указывается вся информация, необходимая для взаимодействия с любыми другими объектами. Реализация скрывает или маскирует от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов (рис. 1.5).

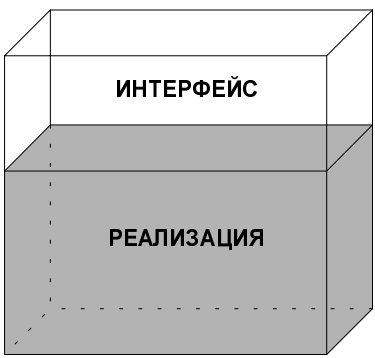


Рис. 1.5. Иллюстрация сокрытия внутренних деталей структуры классов

Примечание

Степень затемнения фона на приведенном выше рисунке имеет более глубокий смысл, чем может показаться на первый взгляд. Если ассоциировать реа-

лизацию программного модуля с водой в аквариуме, то видимость объектов, находящихся в воде, будет зависеть от степени ее чистоты или загрязнения. В ООП существуют различные варианты доступа к свойствам и методам классов, которые получили название *кванторов видимости* свойств и методов. В этом случае использование различных форм видимости для компонентов классов удобно ассоциировать с прозрачностью фона рисунка или видимостью в воде аквариума. Более детальное рассмотрение различных форм видимости приводится в *части II* книги.

Третьим принципом ООП является полиморфизм. Под *полиморфизмом* (греч. poly — много, morfos — форма) понимают свойство некоторых объектов принимать различные внешние формы в зависимости от обстоятельств. Применительно к ООП полиморфизм означает, что действия, выполняемые одноименными методами, могут отличаться в зависимости от того, к какому из классов относится тот или иной метод.

Рассмотрим, например, три объекта соответствующих классов: двигатель автомобиля, электрический свет в комнате и персональный компьютер. Для каждого из них можно определить операцию `выключить()`. Однако результат выполнения этой операции будет отличаться для каждого из рассмотренных объектов. Так, для двигателя автомобиля вызов метода `двигательАвтомобиля.выключить()` означает прекращение подачи топлива и его остановку. Вызов метода `комната.электрическийСвет.выключить()` означает простой щелчок выключателя, после чего комната должна погрузиться в темноту. В последнем случае действие `персональныйКомпьютер.выключить()` может быть причиной потери данных, если выполняется нерегламентированным образом.

Примечание

В рассмотренном выше примере использовалась одна из принятых нотаций в некоторых языках программирования (например, в Object Pascal и C++) для обозначения принадлежности метода тому или иному классу. В соответствии с этой нотацией, сначала указывается имя класса, в котором определен метод, а затем через точку имя самого метода. Если метод определен в некотором подклассе, то должна быть указана вся цепочка классов, начиная с наиболее общего из них. При этом характерным признаком метода является пара скобок, которые используются для указания списка аргументов или формальных параметров данного метода.

Для операции `выключить()`, рассматриваемой в том или ином контексте, можно определить дополнительные параметры, такие как время выключения, некоторое условие нахождения объекта в предварительно включенном состоянии и прочее. С этой целью после имени операции указываются скобки, в которых могут быть перечислены эти дополнительные параметры или аргументы. В случае отсутствия аргументов считается, что список параметров пуст. Однако скобки все равно записываются и указывают на тот факт, что соответствующее имя является именем операции или метода, в отличие от свойств или атрибутов класса, которые записываются без скобок.

Полиморфизм объектно-ориентированных языков связан с *перегрузкой* функций, но не тождествен ей. Важно иметь в виду, что имена методов и свойств тесно связаны с классами, в которых они описаны. Это обстоятельство обеспечивает определенную надежность работы программы, поскольку исключает случайное применение метода для решения несвойственной ему задачи.

Широкое распространение методологии ООП оказало влияние на процесс разработки программ. В частности, процедурно-ориентированная декомпозиция программ уступила место *объектно-ориентированной декомпозиции*, при которой отдельными структурными единицами программы стали являться не процедуры и функции, а классы и объекты с соответствующими свойствами и методами. Как следствие, программа перестала быть последовательностью предопределенных на этапе кодирования действий, а стала *событийно управляемой*. Последнее обстоятельство доминирует и при разработке широкого круга современных приложений. В этом случае каждая программа представляет собой бесконечный цикл ожидания некоторых заранее определенных событий. Инициаторами событий могут быть другие программы или пользователи. При наступлении отдельного события, например, нажатия клавиши на клавиатуре или щелчка кнопкой мыши программа выходит из состояния ожидания и реагирует на это событие вполне адекватным образом. Реакция программы при этом тоже связывается с последующими событиями.

Наиболее существенным обстоятельством в развитии методологии ООП явилось осознание того факта, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы. Действительно, до того, как начать программирование классов, их свойств и методов, необходимо определить, чем же являются сами эти классы. Более того, нужно дать ответы на такие вопросы, как: сколько и какие классы нужно определить для решения поставленной задачи, какие свойства и методы необходимы для придания классам требуемого поведения, а также установить взаимосвязи между классами.

Эта совокупность задач не столько связана с написанием кода, сколько с общим анализом требований к будущей программе, а также с анализом конкретной предметной области, для которой разрабатывается программа. Все эти обстоятельства привели к появлению специальной методологии, получившей название методологии объектно-ориентированного анализа и проектирования (ООАП).

1.3. Методология объектно-ориентированного анализа и проектирования

Необходимость анализа предметной области до начала написания программы была осознана давно при разработке масштабных проектов. Процесс разработки баз данных существенно отличается от написания программного

кода для решения вычислительной задачи. Главное отличие заключается в том, что при проектировании базы данных возникает необходимость в предварительной разработке *концептуальной схемы* или *модели*, которая отражала бы общие взаимосвязи предметной области и особенности организации соответствующей информации. При этом под *предметной областью* принято понимать ту часть реального мира, которая имеет существенное значение или непосредственное отношение к процессу функционирования программы. Другими словами, предметная область включает в себя только те объекты и взаимосвязи между ними, которые необходимы для описания требований и условий решения некоторой задачи.

Выделение исходных или базовых компонентов предметной области, необходимых для решения той или иной задачи, представляет, в общем случае, нетривиальную задачу. Сложность проявляется в неформальном характере процедур или правил, которые можно применять для этой цели. Более того, эта работа должна выполняться совместно со специалистами или экспертами, хорошо знающими предметную область. Например, если разрабатывается база данных для обслуживания пассажиров крупного аэропорта, то в проектировании концептуальной схемы базы данных должны принимать участие штатные сотрудники данного аэропорта. Эти сотрудники должны хорошо знать весь процесс обслуживания пассажиров или данную предметную область.

Для выделения или идентификации компонентов предметной области было предложено несколько способов и правил. Сам этот процесс получил название *концептуализации* предметной области. На предварительном этапе концептуализации конструктивную помощь могут оказать, так называемые, CRC-карточки (Component, Responsibility, Collaborator — компонента, обязанность, сотрудники). При этом под *компонентой* понимают некоторую абстрактную единицу, которая обладает функциональностью, т. е. может выполнять определенные действия, связанные с решением поставленных задач. Для каждой выделенной компоненты предметной области разрабатывается собственная CRC-карточка (рис. 1.6).

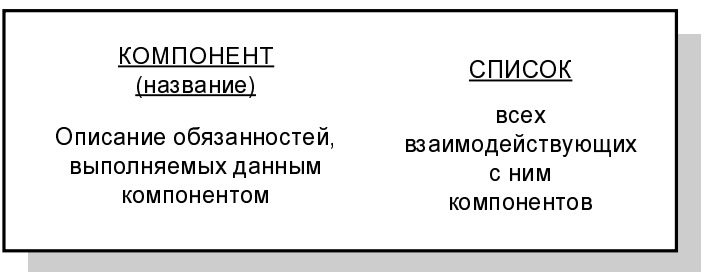


Рис. 1.6. Общий вид CRC-карточки для описания компонентов предметной области

Построение концептуальной модели предметной области с помощью CRC-карточек предполагает выявление и последующую конкретизацию всех компонентов, которые оказывают существенное влияние на решение поставленной задачи. При этом каждый из компонентов служит прототипом некоторого класса при программной реализации соответствующего проекта. Другими словами, отдельные компоненты выбираются таким образом, чтобы при последующей работе над проектом их было удобно представить в форме классов. В этом случае немаловажное значение приобретает и сам способ представления информации о концептуальной схеме предметной области.

Появление методологии ООАП потребовало, с одной стороны, разработки различных средств концептуализации предметной области, а с другой стороны, соответствующих специалистов, которые владели бы этой методологией. Именно на этом этапе появляется необходимость в относительно новом типе специалиста, который получил название *аналитика* или *архитектора*. Наряду со специалистами по предметной области аналитик участвует в построении концептуальной схемы будущей программы, которая затем преобразуется программистами в код.

Разделение процесса разработки сложных программных приложений на отдельные этапы способствовало становлению концепции жизненного цикла программы. Под *жизненным циклом* (ЖЦ) программы понимают совокупность взаимосвязанных и следующих во времени этапов, начиная от разработки требований к ней и заканчивая полным отказом от ее использования. Стандарт ISO/IEC 12207, хотя и описывает общую структуру ЖЦ программы, но не конкретизирует детали выполнения тех или иных этапов. Согласно принятым взглядам ЖЦ программы состоит из следующих этапов:

- анализ предметной области и формулировка требований к программе;
- проектирование структуры программы;
- реализация программы в кодах (собственно программирование);
- внедрение программы;
- сопровождение программы;
- отказ от использования программы.

На этапе анализа предметной области и формулировки требований осуществляется определение функций, которые должна выполнять разрабатываемая программа, а также концептуализация предметной области. Эту работу выполняют аналитики совместно со специалистами предметной области. Результатом является некоторая концептуальная схема, содержащая описание основных компонентов и тех функций, которые они должны выполнять.

Этап проектирования структуры программы заключается в разработке детальной схемы, на которой указываются классы, их свойства и методы, а также различные взаимосвязи между ними. Как правило, на этом этапе могут участвовать в работе аналитики, архитекторы и отдельные квалифи-

цированные программисты. Результатом должна стать детализированная схема программы, на которой указываются все классы и взаимосвязи между ними в процессе функционирования программы. Согласно методологии ООАП, данная схема должна предшествовать этапу написания программного кода.

Этап программирования вряд ли нуждается в уточнении, поскольку является наиболее традиционным для программистов. Появление инструментариев *быстрой разработки приложений* (Rapid Application Development, RAD) позволило существенно сократить время и затраты на выполнение данного этапа. Результатом данного этапа является программное приложение, которое обладает требуемой функциональностью и способно решать нужные задачи в конкретной предметной области.

Этапы внедрения и сопровождения программы связаны с необходимостью настройки и конфигурирования среды программы, а также с устранением возникших в процессе ее использования ошибок. Иногда в качестве отдельного этапа выделяют *тестирование* программы, под которым понимают проверку работоспособности программы на некоторой совокупности исходных данных или при некоторых специальных режимах эксплуатации. Результатом этих этапов является повышение надежности приложения, исключающего возникновение критических ситуаций или нанесение ущерба компании, использующей его.

Примечание

Рассматривая различные этапы ЖЦ программы, следует отметить одно важное обстоятельство. А именно, если появление RAD-инструментариев позволило существенно сократить сроки этапа программирования, то отсутствие соответствующих средств для первых двух этапов долгое время сдерживало процесс разработки приложений. Развитие методологии ООАП было направлено на автоматизацию второго, а затем и первого этапов ЖЦ программы.

Методология ООАП тесно связана с концепцией *автоматизированной разработки программного обеспечения* (Computer Aided Software Engineering, CASE). Появление первых CASE-средств было встречено с определенной настороженностью. Со временем появились как восторженные отзывы об их применении, так и критические оценки их возможностей. Причин для столь противоречивых мнений было несколько. Первая из них заключается в том, что ранние CASE-средства были простой надстройкой над некоторой системой управления базами данных (СУБД). Хотя визуализация процесса разработки концептуальной схемы базы данных (БД) имеет немаловажное значение, но она не решает проблем разработки приложений других типов.

Вторая причина имеет более сложную природу, поскольку связана с графической нотацией, реализованной в том или ином CASE-средстве. Если языки программирования имеют строгий синтаксис, то попытки предложить подходящий синтаксис для визуального представления концептуальных схем

БД были восприняты далеко не однозначно. Появилось несколько подходов, которые будут более подробно рассмотрены далее *(см. главу 2)*. На этом фоне разработка и стандартизация *унифицированного языка моделирования* (Unified Modeling Language, UML), ориентированного на решение задач первых двух этапов ЖЦ программ, было воспринято с большим оптимизмом всем сообществом корпоративных программистов.

Последнее, на что следует обратить внимание, это осознание необходимости построения предварительной модели программной системы, которую, согласно современным концепциям ООАП, следует считать результатом первых этапов ЖЦ программы. Поскольку язык UML даже в своем названии имеет отношение к моделированию, следует дополнительно остановиться на целом ряде достаточно важных вопросов. Таким образом, мы переходим к теме, которая традиционно не рассматривается в изданиях по ООАП, но имеет самое прямое отношение к процессу построения моделей и собственно моделированию. Речь идет о методологии системного анализа и системного моделирования.

1.4. Методология системного анализа и системного моделирования

Системный анализ как научное направление имеет более давнюю историю, чем ООП и ООАП, и собственный предмет исследования. Центральным понятием системного анализа является понятие *системы*, под которой понимается совокупность объектов, компонентов или элементов произвольной природы, образующих некоторую целостность. Определяющей предпосылкой выделения некоторой совокупности как системы является возникновение у нее новых свойств или характеристик, которых не имеют составляющие ее элементы. Примерами систем являются: персональный компьютер, автомобиль, человек, биосфера, программа и многие другие. Более ортодоксальная точка зрения утверждает, что все окружающие нас предметы являются системами.

Важнейшими характеристиками любой системы являются ее структура и процесс функционирования. Под *структурой системы* понимают устойчивую во времени совокупность взаимосвязей между ее элементами или компонентами. Именно структура связывает воедино все элементы и препятствует распаду системы на отдельные составляющие ее элементы. Структура системы может отражать самые различные взаимосвязи, в том числе, и вложенность элементов одной системы в другую. В этом случае принято называть более мелкую или вложенную систему *подсистемой*, а более крупную — *метасистемой*.

Процесс функционирования системы тесно связан с изменением ее свойств или поведения во времени. При этом важной характеристикой системы

является ее *состояние*, под которым понимается совокупность свойств или признаков, которые в каждый момент отражают наиболее существенные особенности поведения системы.

Рассмотрим в качестве примера систем — "Автомобиль" и "Двигатель". С одной стороны, двигатель является составной частью автомобиля или элементом системы "Автомобиль". С другой стороны, двигатель сам является системой, состоящей из отдельных компонентов, таких как цилиндры, свечи зажигания и другие. Поэтому система "Двигатель" также будет являться подсистемой системы "Автомобиль". В свою очередь для системы "Двигатель" система охлаждения двигателя будет являться подсистемой.

Структура системы "Автомобиль" может быть описана с разных точек зрения. Наиболее общее представление о структуре этой системы дает механическая схема устройства того или иного автомобиля. Взаимодействие элементов в этом случае носит механический характер. Состояние автомобиля можно рассматривать также с различных точек зрения, наиболее общей из которых является характеристика автомобиля как исправного или неисправного. Очевидно, что каждое из этих состояний в отдельных ситуациях может быть детализировано. Например, состояние "неисправный" может быть конкретизировано в состояния "неисправность двигателя", "неисправность аккумулятора", "отсутствие подачи топлива" и прочее. Важно иметь четкое представление, что подобная детализация должна быть адекватна решаемой задаче.

Процесс функционирования системы отражает поведение системы во времени и может быть представлен как последовательное изменение ее состояний. Если система изменяет одно свое состояние на другое состояние, то принято говорить, что система *переходит* из одного состояния в другое. Совокупность признаков или условий изменения состояний системы в этом случае называется *переходом*. Для системы с дискретными состояниями процесс функционирования может быть представлен в виде последовательности состояний с соответствующими переходами. Более точное графическое описание процесса функционирования систем с использованием нотации языка UML будет дано в *главе 8*.

Методология системного анализа служит концептуальной основой для системно-ориентированной декомпозиции предметной области. В этом случае исходными компонентами концептуализации являются системы и взаимосвязи между ними. При этом понятие системы является более общим, чем понятия классов и объектов в ООАП. Результатом системного анализа является построение некоторой модели системы или предметной области.

Примечание

Понятие модели столь широко используется в повседневной жизни, что приобрело очень много смысловых оттенков. Это и "Дом моделей" известного кутюрье,

и модель престижной марки автомобиля, и модель политического руководства, и математическая модель колебаний маятника. Применительно к программным системам нас будет интересовать только то понятие модели, которое используется в системном анализе.

В системном анализе под *моделью* понимается некоторое представление о системе, отражающее наиболее существенные закономерности ее структуры и процесса функционирования и зафиксированное на некотором языке или в другой форме. Примерами моделей могут служить: аэродинамическая модель гоночного автомобиля и проектируемого самолета, модель ракетного двигателя, модель колебательной системы, модель системы электроснабжения региона, модель избирательной компании и другие.

Общим свойством всех моделей является их подобие оригинальной системе или системе-оригиналу. Важность построения моделей заключается в возможности их использования для получения информации о свойствах или поведении системы-оригинала. При этом процесс построения и последующего применения моделей для получения информации о системе-оригинале получил название *моделирование*.

Примечание

Аналогично высказанному ранее замечанию, термин "моделирование" также имеет довольно много смысловых оттенков, например, моделирование одежды или моделирование прически. Не отрицая важности этих сфер творчества, следует отметить, что все эти аспекты моделирования лежат за пределами книги. Рассмотрение особенностей языка UML связано с вопросами логического или информационного моделирования систем.

Наиболее общей моделью системы является так называемая модель "черного ящика". В этом случае система представляется в виде прямоугольника, внутреннее устройство которого либо скрыто от аналитика, либо неизвестно. Однако система не является полностью изолированной от внешней среды, поскольку последняя оказывает на систему некоторые информационные или материальные воздействия. Такие воздействия получили название *входных воздействий*. В свою очередь, система также оказывает на среду или другие системы определенные информационные или материальные воздействия, которые получили название *выходных воздействий*. Графически данная модель может быть изображена следующим образом (рис. 1.7).

Ценность моделей, подобных модели "черного ящика", весьма условна. Невольно может возникнуть ассоциация с "Черным квадратом". Однако если оценка изобразительных особенностей последнего не входит в задачи системного анализа, то общая модель содержит некоторую важную информацию о функциональных особенностях данной системы, которые дают представление о поведении. Действительно, кроме самой общей информации о том, на какие воздействия реагирует система и как проявляется эта реакция

на окружающие объекты и системы, другой информации мы получить не можем. В рамках системного анализа разработаны определенные методологические средства, позволяющие выполнить дальнейшую конкретизацию общей модели системы. Некоторые из графических средств представления моделей систем будут рассмотрены в *главе 2*.

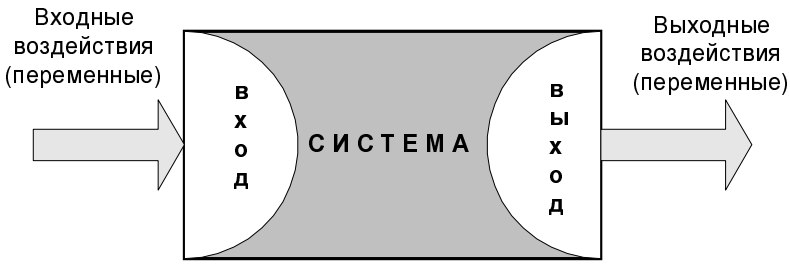


Рис. 1.7. Графическое изображение модели системы в виде "черного ящика"

Процесс разработки адекватных моделей и их последующего конструктивного применения требует не только знания общей методологии системного анализа, но и наличия соответствующих изобразительных средств или языков для фиксации результатов моделирования и их документирования. Очевидно, что текст на том или ином языке не вполне подходит для этой цели, поскольку обладает нечеткостью, неоднозначностью и неопределенностью. В то же время для построения и фиксации знаний о моделях были разработаны различные теоретические методы, основанные на развитии математических и формально-логических средств моделирования, а также предложены различные графические нотации, отражающие специфику решаемых задач. Важно представлять, что унификация любого языка моделирования тесно связана с методологией системного моделирования, т. е. с системой воззрений и принципов рассмотрения сложных явлений и объектов как моделей сложных систем.

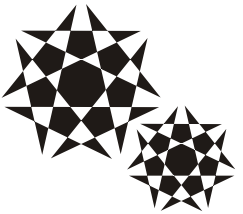
Сложность системы и, соответственно, ее модели может быть рассмотрена с различных точек зрения. Прежде всего, можно выделить сложность структуры системы, которая характеризуется количеством элементов системы и различными типами взаимосвязей между этими элементами. Если количество элементов превышает некоторое пороговое значение, которое, вообще говоря, не является фиксированным, то такая система может быть названа сложной. Например, если программная СУБД насчитывает более 100 отдельных форм ввода и вывода информации, то многие программисты сочтут ее сложной. Транспортная система современных мегаполисов также может служить примером сложной системы.

Вторым аспектом сложности является сложность процесса функционирования системы. Это может быть связано как с непредсказуемым характером

поведения системы, так и невозможностью формального представления правил преобразования входных воздействий в выходные. В качестве примеров сложных программных систем можно привести современные операционные системы, которым присущи черты сложности, как структуры, так и поведения.

В завершение этой главы следует отметить, что разработка и последующая унификация графических средств моделирования программных систем тесно связана с необходимостью построения концептуальных моделей именно сложных систем, которые являются предметом выполнения реальных проектов. Действительно, если модель простой системы может быть объяснена собеседнику в "двух словах", то потенциальная возможность построения исчерпывающей модели сложной системы может отсутствовать в принципе. Выходом из создавшегося положения является декомпозиция или разбиение исходной системы на отдельные ее подсистемы с последующей разработкой моделей подсистем. В этом контексте важную роль играют различные представления модели систем в форме канонических диаграмм языка UML.

Глава 2



Исторический обзор развития методологии объектно-ориентированного анализа и проектирования сложных систем

2.1. Предыстория. Математические основы

Представление знаний о различных объектах и системах окружающего нас мира при помощи графической символики уходит своими истоками в глубокую древность. В качестве примеров можно привести условные обозначения знаков Зодиака, магические символы различных оккультных доктрин, графические изображения геометрических фигур на плоскости и в пространстве. Важным достоинством той или иной графической нотации является возможность образного закрепления содержательного смысла или семантики отдельных понятий, что существенно упрощает процесс общения между посвященными в соответствующие теории и идеологии.

2.1.1. Теория множеств

Как одну из наиболее известных систем графических символов, оказавших непосредственное влияние на развитие научного мышления, следует отметить язык диаграмм английского логика Джона Венна (1834—1923). В настоящее время *диаграммы Венна* применяются для иллюстрации основных теоретико-множественных операций, которые являются предметом специального раздела математики — *теории множеств*. Поскольку многие общие идеи моделирования систем имеют адекватное описание в терминологии теории множеств, рассмотрим основные понятия данной теории, имеющие отношение к современным концепциям и технологиям исследования сложных систем.

Исходным понятием теории множеств является само понятие *множество*, под которым принято понимать некоторую совокупность объектов, хорошо различимых нашей мыслью или интуицией. При этом не делается никаких

предположений ни о природе этих объектов, ни о способе их включения в данную совокупность. Отдельные объекты, составляющие то или иное множество, называют *элементами* данного множества. Вопрос "Почему мы рассматриваем ту или иную совокупность элементов как множество?" не требует ответа, поскольку в общее определение множества не входят никакие дополнительные условия на включение отдельных элементов в множество. Если нам хочется, например, рассмотреть множество, состоящее из трех элементов: солнце, море, апельсин, то никто не сможет запретить это сделать.

Примерами множеств являются: множество квартир жилого дома, множество книг в библиотеке, множество натуральных чисел, совокупность компьютеров в офисе, штат сотрудников фирмы, множество живущих на планете людей, множество звезд на небосводе.

Примечание

Создается впечатление, что ситуация с заданием множеств более или менее очевидна. Но это впечатление обманчиво. Даже не говоря об известных парадоксах теории множеств, как быть с множеством мыслей отдельного человека? Или множеством всех красок, которые встречаются в природе? Однако такие каверзные случаи мы рассматривать не будем, ограничив круг ситуаций такими, в которых идентификация отдельных элементов множеств не превращается в серьезную проблему. С другой стороны, процесс моделирования сложных систем зачастую сопряжен именно с подобного рода трудностями.

В теории множеств используется специальное соглашение, по которому множества обозначаются прописными буквами латинского алфавита, и традиция эта настолько общепризнанна, что не возникает никакого сомнения в ее целесообразности. При этом отдельные элементы обозначаются строчными буквами, иногда с индексами, которые вносят некоторую упорядоченность в последовательность рассмотрения этих элементов. Важно понимать, что какой бы то ни было порядок, вообще говоря, не входит в исходное определение множества.

Таким образом, множество, например, квартир 100-квартирного жилого дома с использованием специальных обозначений можно записать следующим образом: $A = \{a_1, a_2, a_3, ..., a_{100}\}$. Здесь фигурные скобки служат обозначением совокупности элементов, каждый из которых имеет свой уникальный числовой индекс. Важно понимать, что для данного конкретного множества элемент a_{10} обозначает отдельную квартиру в рассматриваемом жилом доме. При этом вовсе не обязательно, чтобы номер этой квартиры был равен 10, хотя с точки зрения удобства это было бы желательно.

Примечание

При выборе обозначений для множеств допускается некоторый произвол, который не всегда понятен лицам, далеким от математики. Однако здесь уместна

аналогия с выбором имен для переменных и процедур в языках высокого уровня, когда программист сам решает, как ему обозначать соответствующую конструкцию в программе.

Принято называть элементы отдельного множества *принадлежащими* данному множеству. Данный факт записывается при помощи специального символа "$\in$", который так и называется — *символ принадлежности*. Например, запись $a_{10} \in A$ означает тот простой факт, что отдельная квартира (возможно, с номером 10) принадлежит рассматриваемому множеству квартир некоторого жилого дома.

Следующим важным понятием, которое служит прототипом многих более конкретных терминов при моделировании сложных систем, является понятие *подмножества*. Казалось бы, интуитивно и здесь нет ничего неясного. Если есть некоторая совокупность, рассматриваемая как множество, то любая ее часть и будет являться подмножеством. Так, например, совокупность квартир на первом этаже жилого дома есть не что иное, как подмножество рассматриваемого нами примера. Ситуация становится не столь тривиальной, если рассматривать множество абстрактных понятий, таких как сущность или класс.

Для обозначения подмножества используется специальный символ. Если утверждается, что множество A является подмножеством множества B, то это записывается как $A \subset B$. Запоминать подобные значки не всегда удобно, поэтому со временем была предложена специальная система графических обозначений.

Как же используются диаграммы Венна в теории множеств? Тот факт, что некоторая совокупность элементов образует множество, можно обозначить графически в виде круга. В этом случае окружность имеет содержательный смысл или, выражаясь более точным языком, семантику *границы* данного множества. Тогда вполне логично отношение включения элементов одного множества в другое изобразить графически следующим образом (рис. 2.1). На этом рисунке большему множеству B соответствует внешний круг, а меньшему множеству (подмножеству) A — внутренний.

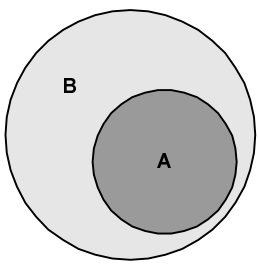


Рис. 2.1. Диаграмма Венна для отношения включения двух множеств

Подобным образом можно изобразить и основные теоретико-множественные операции. Так, в общем случае, *пересечением* двух множеств A и B называется некоторое третье множество C, которое состоит из тех элементов двух исходных множеств, которые *одновременно* принадлежат и множеству A, и множеству B. Для этой операции также имеется специальное обозначение: $C = A \cap B$. Например, если для операции пересечения в качестве множества A рассмотреть множество сотрудников некоторой фирмы, а в качестве множества B — множество всех мужчин, то нетрудно догадаться, что множество C будет состоять из элементов — всех сотрудников мужского пола данной фирмы. Операция пересечения множеств также может быть проиллюстрирована с помощью диаграмм Венна (рис. 2.2). На этом рисунке условно изображены два множества A и B, а затененной области как раз и соответствует множество C, являющееся пересечением множеств A и B.

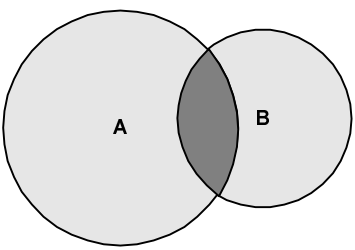


Рис. 2.2. Диаграмма Венна для пересечения двух множеств

Следующей операцией, которая также допускает наглядную интерпретацию, является *объединение* множеств. Под объединением двух множеств A и B понимается некоторое третье, пусть это будет множество D, которое состоит из тех элементов, которые принадлежат или A, или B, или им обоим одновременно. Конечно, специальное обозначение есть и для этой операции: $D = A \cup B$. Так, если в качестве A рассмотреть множество, состоящее из клавиатуры и мыши, а в качестве B — множество, состоящее из системного блока и монитора, то нетрудно догадаться, что их объединение, т. е. D, образует основные составляющие персонального компьютера. Эта операция также может быть условно изображена графически (рис. 2.3). На этом рисунке объединению соответствует затемненная область, только размеры и форма ее отличаются от случая пересечения на предыдущем рисунке.

Хотя существуют и другие операции над множествами, а также целый ряд с ними связанных дополнительных понятий, их рассмотрение выходит за рамки настоящей книги. Последнее, на что следовало бы обратить внимание при столь кратком знакомстве с основами теории множеств — это на так называемые понятия *мощности* множества и *отношения* множеств. Что касается понятия мощности, то данный термин важен для анализа кратности связей, поскольку ассоциируется с количеством элементов отдельного мно-

жества. В случае конечного множества ситуация очень простая, поскольку мощность конечного множества равна количеству его элементов. Таким образом, возвращаясь к примеру с множеством *A* квартир жилого дома, можно сказать, что его мощность равна 100.

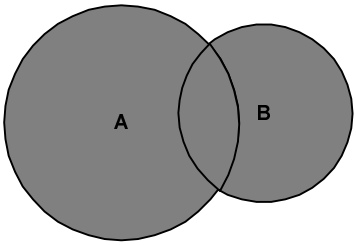


Рис. 2.3. Диаграмма Венна для объединения двух множеств

Ситуация усложняется, когда рассматриваются бесконечные множества. Не вдаваясь в технические детали, которые послужили источником драматичного по своим последствиям кризиса основ математики, ограничим наше рассмотрение бесконечными множествами *счетной* мощности. Ими принято считать множества, содержащие бесконечное число элементов, которые, однако, можно перенумеровать натуральными числами 1, 2, 3 и т. д. При этом важно иметь в виду, что достичь последнего элемента при такой нумерации принципиально невозможно, иначе множество окажется конечным.

Например, есть все основания считать множество всех звезд бесконечным, хотя многие из них имеют свое уникальное название. С другой стороны, множество всех возможных комбинаций из 8 символов, которые могут служить для ввода некоторого пароля, конечное, хотя и достаточно большое. Точнее, это множество имеет конечную мощность.

Примечание

Проблема бесконечного могла бы показаться отвлеченной и имеющей некоторый философский оттенок, если бы не ее связь с моделированием сложных систем. Так, при рассмотрении некоторой предметной области с целью построения ее модели приходится выделять конечное число сущностей, образующих определенный "скелет" или остов архитектуры будущей модели. И это при том, что реальность предметов допускает бесконечное рассмотрение их свойств, атрибутов и взаимосвязей.

Наконец, было упомянуто и следующее понятие, различные аспекты которого будут служить темой рассмотрения во всех последующих главах. Это фундаментальное понятие *отношения* множеств, которое часто заменяется терминами *связь* или *соотношение*. Этот термин ведет свое происхождение от теории множеств и служит для обозначения любого подмножества упорядоченных кортежей, построенных из элементов некоторых исходных

множеств. При этом под *кортежем* понимается просто набор или список элементов, важно только, чтобы они были упорядочены. Другими словами, если рассматривать первый элемент кортежа, то он всегда будет первым в списке элементов, второй элемент кортежа будет вторым элементом в списке и т. д. Можно ли это записать с использованием специальных обозначений?

Хотя и существует некоторая неоднозначность в принятых обозначениях, кортеж из двух элементов удобно обозначать как $<a_1, a_2>$, из трех — $<a_1, a_2, a_3>$ и т. д. При этом отдельные элементы могут принадлежать как одному и тому же, так и различным множествам. Важно иметь в виду, что порядок выбора элементов для построения кортежей строго фиксирован для конкретной задачи. Речь идет о том, что первый элемент всегда выбирается из первого множества, второй — из второго и т. д. Отношение в этом случае будет характеризовать способ или семантику выбора отдельных элементов из одного или нескольких множеств для подобного упорядоченного списка. В этом смысле взаимосвязь является частным случаем отношения, о чем будет сказано в последующем.

К сожалению, диаграммы Венна не предназначены для иллюстрации отношений в общем случае. Однако отношения послужили исходной идеей для развития другой теории, которая даже в своем названии несет отпечаток графической нотации, а именно — теории графов. В этой связи наиболее важным является тот факт, что теоретико-множественные отношения послужили также основой для разработки реляционной алгебры в теории реляционных баз данных. Развитие последней привело к тому, что в последние годы именно реляционные СУБД конкретных фирм доминируют на рынке соответствующего программного обеспечения.

2.1.2. Теория графов

Граф можно рассматривать как графическую нотацию для бинарного отношения двух множеств. Бинарное отношение состоит из таких кортежей или списков элементов, которые содержат только два элемента некоторого множества. Хотя основные понятия теории графов получили свое развитие задолго до появления теории множеств как самостоятельной научной дисциплины, формальное определение графа удобно представить в теоретико-множественных терминах.

Графом называется совокупность двух множеств: множества точек (или *вершин*) и множества соединяющих их линий (или *ребер*). Формально граф задается в виде двух множеств: $G = (V, E)$, где $V = \{v_1, v_2, ..., v_n\}$ — множество вершин графа, *а* $E = \{e_1, e_2, ..., e_m\}$ — множество ребер графа. Натуральное число n определяет общее количество вершин конкретного графа, а натуральное число m — общее количество ребер графа. Следует заметить, что в общем случае не все вершины графа могут соединяться между собой, что ставит в соответствие каждому графу некоторое бинарное отношение P_G,

состоящее из всех пар вида $<v_i, v_j>$, где $v_i, v_j \in V$. При этом пара $<v_i, v_j>$ и, соответственно, пара $<v_j, v_i>$ принадлежат отношению P_G в том случае, если вершины v_i и v_j соединяются в графе G некоторым ребром $e_k \in E$. Вершины графа изображаются точками, а ребра — отрезками прямых линий. Рядом с вершинами и ребрами записываются соответствующие номера (или идентификаторы), позволяющие идентифицировать их однозначным образом.

Вообще говоря, графы бывают двух различных типов. Рассмотренное выше определение относится к *неориентированному* графу, т. е. к такому графу, у которого связывающие вершины ребра не имеют направления или ориентации. Кроме неориентированных графов существуют *ориентированные* графы, которые определяются следующим образом.

Ориентированный граф также задается в виде двух множеств $G = (V, E)$, где $V = \{v_1, v_2, ..., v_n\}$ — множество вершин графа, а $E = \{e_1, e_2, ..., e_m\}$ — множество *дуг* графа. Натуральное число n определяет общее количество вершин конкретного графа, а натуральное число m — общее количество дуг графа. При этом каждая дуга $e_k \in E$ ориентированного графа G имеет свое начало — некоторую единственную вершину $v_i \in V$ и конец — некоторую единственную вершину $v_j \in V$. В отличие от ребра, дуга всегда имеет обозначение со стрелочкой, которая направлена к конечной вершине дуги. Множество дуг ставит в соответствие каждому ориентированному графу некоторое бинарное отношение P_G, состоящее из всех пар вида $<v_i, v_j>$, где $v_i, v_j \in V$. При этом пара $<v_i, v_j>$ принадлежит отношению P_G в том случае, если вершины v_i и v_j соединяются в графе G некоторой дугой $e_k \in E$ с началом в вершине v_i и концом в вершине v_j.

Ниже представлены два примера конкретных графов (рис. 2.4). При этом первый из них (рис. 2.4, *а*) является неориентированным графом, а второй (рис. 2.4, *б*) — ориентированным графом. Как нетрудно заметить, для неориентированного графа ребро e_1 соединяет вершины v_1 и v_2, ребро e_2 соединяет вершины v_1 и v_3, ребро e_3 соединяет вершины v_2 и v_3 и так далее. Последнее ребро, e_8, соединяет вершины v_4 и v_5, тем самым задается описание графа в целом. Других ребер данный граф не содержит, также как не содержит других вершин, не изображенных на рисунке. Так, хотя ребра e_6 и e_7 визуально пересекаются, но точка их пересечения не является вершиной графа.

Для ориентированного графа (рис. 2.4, *б*) ситуация несколько иная. А именно, вершины v_1 и v_2 соединены дугой e_1, для которой вершина v_2 является началом дуги, а вершина v_1 — концом этой дуги. Далее дуга e_2 соединяет вершины v_1 и v_4, при этом началом дуги e_2 является вершина v_1, а концом — вершина v_4.

Графы широко применяются для представления различной информации о структуре систем и процессов. Примерами подобных графических моделей

могут служить: схемы автомобильных дорог, соединяющих отдельные населенные пункты; схемы телекоммуникаций, используемых для передачи информации между отдельными узлами; схемы программ, на которых указываются варианты ветвления вычислительного процесса. Общим свойством подобных моделей является возможность представления информации в графическом виде в форме соответствующего графа. При этом отдельные модели, как правило, обладают дополнительной семантикой и специальными обозначениями, характерными для той или иной предметной области.

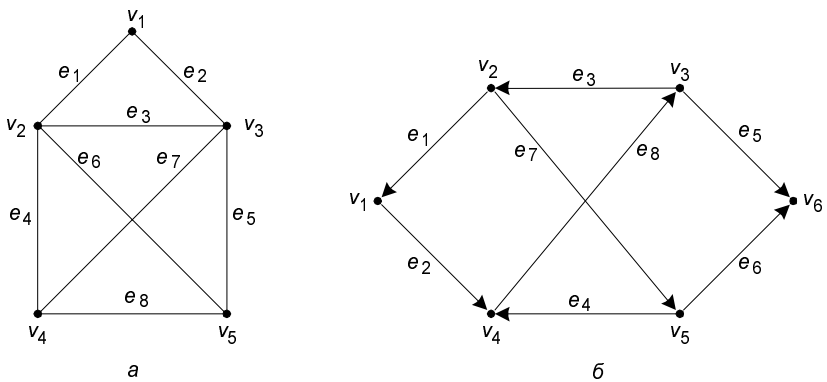


Рис. 2.4. Примеры неориентированного (*а*) и ориентированного (*б*) графов

Важными понятиями теории графов являются понятия маршрута и пути, которые ассоциируются с последовательным перемещением от вершины к вершине по соединяющим их ребрам или дугам. Для неориентированного графа *маршрут* определяется как конечная или бесконечная упорядоченная последовательность ребер $S = <..., e_{s1}, e_{s2}, ..., e_{sk},>$, таких, что каждые два соседних ребра имеют общую вершину. Нас будут интересовать только конечные маршруты $S = <e_{s1}, e_{s2}, ..., e_{sk}>$, т. е. такие, которые состоят из конечного числа ребер. При этом ребро e_{s1} принято считать началом маршрута S, а ребро e_{sk} — концом маршрута S. Для ориентированного графа соответствующая последовательность дуг $S = <e_{s1}, e_{s2}, ..., e_{sk}>$ называется *ориентированным маршрутом*, если две соседние дуги имеют общую вершину, которая является концом предыдущей и началом последующей дуги.

Примерами маршрутов для неориентированного графа (рис. 2.4, *а*) являются последовательности ребер: $S_1 = <e_1, e_2, e_5, e_8>$, $S_2 = <e_1, e_2, e_3, e_1>$, $S_3 = <e_3, e_5, e_8>$. Если в маршруте не повторяются ни ребра, ни вершины, как в случае S_1 и S_3, то такой неориентированный маршрут называется *простой цепью*.

Примерами ориентированных маршрутов для графа (рис. 2.4, *б*) являются такие последовательности дуг: $S_1 = <e_2, e_8, e_5>$, $S_2 = <e_3, e_7, e_6>$,

$S_3 = <e_8, e_3, e_7, e_4, e_8>$. Если в ориентированном маршруте не повторяются ни ребра, ни вершины, как в случае S_1 и S_2, то такой ориентированный маршрут называется *путем*. Последнее понятие также иногда применяется для обозначения простой цепи в неориентированных графах и для определения специального класса графов, так называемых деревьев. В общем случае деревья служат для графического представления иерархических структур или иерархий, занимающих важное место в ООАП.

Деревом в теории графов называется такой граф $D = <V, E>$, между любыми двумя вершинами которого существует единственная простая цепь, т. е. неориентированный маршрут, у которого вершины и ребра не повторяются. Применительно к ориентированным графам, соответствующее определение является более сложным, поскольку основывается на выделении некоторой специальной вершины v_0, которая получила специальное название *корневой* вершины или просто — *корня*. В этом случае ориентированный граф $D = <V, E>$ называется ориентированным деревом или сокращенно — деревом, если между корнем дерева v_0 и любой другой вершиной существует единственный путь, берущий начало в v_0. Ниже представлены два примера деревьев: неориентированного (рис. 2.5, *а*) и ориентированного (рис. 2.5, *б*).

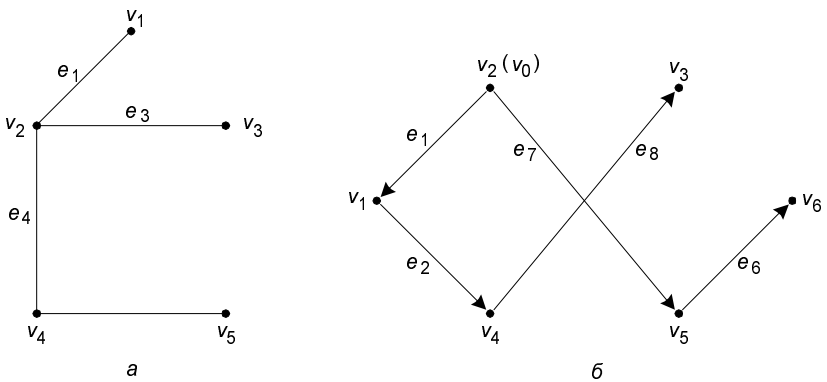


Рис. 2.5. Примеры неориентированного (*а*) и ориентированного (*б*) деревьев

В случае неориентированного дерева (рис. 2.5, *а*) любая из вершин графа может быть выбрана в качестве корня. Подобный выбор определяется специфическими особенностями решаемой задачи. Так, вершина v_1 может рассматриваться в качестве корня неориентированного дерева, поскольку между v_1 и любой другой вершиной дерева всегда существует единственная простая цепь по определению (или, что менее строго, единственный неориентированный путь).

Для случая ориентированного дерева (рис. 2.5, *б*) вершина v_2 является единственным его корнем и имеет специальное обозначение v_0. Единственность

корня в ориентированном дереве следует из того факта, что ориентированный путь всегда имеет единственную вершину, которая является его началом. Поскольку в теории графов имеет значение только наличие или отсутствие связей между отдельными вершинами, деревья, как правило, изображаются специальным образом в виде иерархической структуры. При этом корень дерева изображается самой верхней вершиной в данной иерархии. Далее следуют вершины уровня 1, которые связаны с корнем одним ребром или одной дугой. Следующий уровень будет иметь номер 2, поскольку соответствующие вершины должны быть связаны с корнем двумя последовательными ребрами или дугами. Процесс построения иерархического дерева продолжается до тех пор, пока не будут рассмотрены вершины, не связанные с иными, кроме рассмотренных, или из которых не выходит ни одна дуга. В этом случае самые нижние вершины иногда называют *листьями* дерева. Важно иметь в виду, что в теории графов дерево "растет" вниз, а не вверх.

Изображенные выше деревья (рис. 2.5) можно преобразовать к виду иерархий. Например, неориентированное дерево (рис. 2.5, *а*) может быть представлено в виде иерархического дерева следующим образом (рис. 2.6, *а*). В этом случае корнем иерархии является вершина v_1. Ориентированное дерево (рис. 2.5, *б*) также может быть изображено в форме иерархического дерева (рис. 2.6, *б*), однако такое представление является единственным.

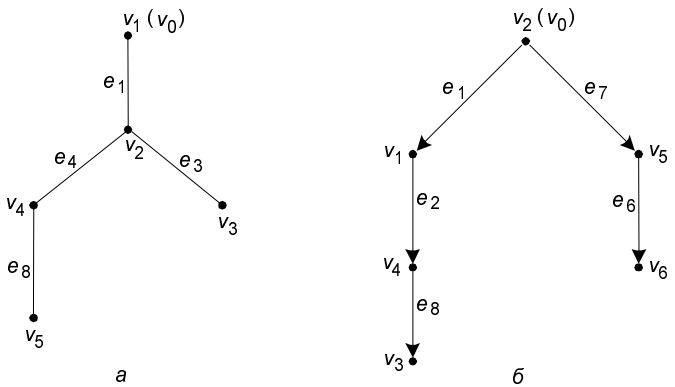


Рис. 2.6. Иерархические схемы неориентированного дерева (*а*) и ориентированного дерева (*б*)

В первом случае (рис. 2.6, *а*) вершина v_2 образует первый уровень иерархии, v_4 и v_3 — второй уровень иерархии, v_5 — третий и последний уровень иерархии. При этом листьями данного неориентированного дерева являются вершины v_3 и v_5. Во втором случае (рис. 2.6, *б*) вершины v_1 и v_5 образуют первый уровень иерархии, v_4 и v_6 — второй уровень иерархии, v_3 — третий и последний уровень иерархии. Листьями данного ориентированного дерева являются вершины v_3 и v_6.

В заключение следует заметить, что в теории графов разработаны различные методы анализа отдельных классов графов и алгоритмы построения специальных графических объектов, рассмотрение которых выходит за рамки данной книги. Для получения дополнительной информации по данной теме можно рекомендовать обратиться к специальной литературе по теории графов, где данные вопросы рассмотрены более подробно. В дальнейшем нас будет интересовать отдельное направление в теории графов, которое связано с явным включением семантики в традиционные обозначения и получившее самостоятельное развитие в форме семантических сетей.

2.1.3. Семантические сети

Семантические сети получили свое развитие в рамках научного направления, связанного с представлением знаний для моделирования рассуждений человека. Эта область научных исследований возникла в рамках общей проблематики искусственного интеллекта и была ориентирована на разработку специальных языков и графических средств для представления декларативных или, что менее точно, статических знаний о предметной области. Результаты исследований, в области семантических сетей, в последующем были конкретизированы и успешно использованы при построении концептуальных моделей и схем реляционных баз данных.

В общем случае под *семантической сетью* понимают некоторый граф $G_s = (V_s, E_s)$, в котором множества вершин V_s и ребер E_s разделены на отдельные типы, обладающие специальной семантикой, характерной для той или иной предметной области. В данной ситуации множество вершин может соответствовать объектам или сущностям рассматриваемой предметной области, и иметь вместо номеров вершин соответствующие явные имена этих сущностей. Подобные имена должны позволять однозначно идентифицировать соответствующие объекты, при этом общих формальных правил записи имен не существует. Множество ребер также делится на различные типы, которые соответствуют различным видам связей между сущностями рассматриваемой предметной области.

Так, при построении семантической сети для представления знаний о рабочем персонале некоторой компании, в качестве объектов целесообразно выбрать отдельных сотрудников, каждого из которых идентифицировать собственным именем и фамилией. Дополнительно, в сети могут присутствовать такие объекты, как рабочие проекты и подразделения компании. В качестве семантических связей можно выделить такие виды, как должностное подчинение сотрудников, их участие в работе над проектами, принадлежность сотрудников тому или иному подразделению компании.

Важной особенностью семантических сетей является разработка специальных графических обозначений для представления отдельных типов вершин и ребер. При этом вершины не изображаются, как ранее — точками, а имеют

вид прямоугольников, овалов, окружностей и других геометрических фигур, конкретный вид которых определяет тот или иной тип сущностей предметной области. Более разнообразным становится и изображение ребер, приобретающих вид различных линий со стрелками или без них, а также имеющих специальные обозначения или украшения в виде условных значков. Соответствующая система обозначений, предназначенная для представления информации об отдельных аспектах моделируемой предметной области, получила название *графической нотации*.

Примечание

В этой связи следует заметить, что различные виды диаграмм языка UML в общем случае являются специальными классами семантических сетей с достаточно развитой семантикой используемых условных обозначений. При этом унифицированный характер этих обозначений определяет конструктивность их использования для моделирования широкого круга приложений.

В качестве конкретного варианта представления информации в виде семантической сети рассмотрим дальнейшее развитие примера с классом "Автомобиль" из *главы 1*. Фрагмент семантической сети, которая описывает иерархию классов данной предметной области, может быть изображен следующим образом (рис. 2.7). На данном рисунке отдельные вершины семантической сети изображаются прямоугольниками с закругленными концами и служат для условного обозначения классов данной предметной области. Соединяющие вершины ребра имеют вполне определенный смысл или семантику. А именно, они явно указывают, что вершина или класс, расположенные на рисунке ниже, являются подклассом того класса уровнем выше, с которым имеется связь в форме соединяющего их ребра.

Например, классы "Легковой автомобиль" и "Грузовой автомобиль" являются подклассами класса "Автомобиль", а "Модель ВАЗ-21083" и "Модель ВАЗ-21099" являются подклассами класса "Легковой автомобиль производства ВАЗ". Ребра или связи данной семантической сети имеют единственный тип, определяемый семантикой включения классов друг в друга. Поэтому никаких дополнительных обозначений они не содержат.

Примечание

Изображенный выше фрагмент семантической сети может быть расширен различным образом, что определяется спецификой решаемой задачи. С одной стороны, можно ввести в рассмотрение дополнительные модели автомобилей, а с другой — другие типы объектов, например, конкретные заводы, расположенные в различных регионах, или станции, осуществляющие техническое обслуживание автомобилей. В последнем случае появляются дополнительные связи, которые могут соответствовать совершенно иной семантике. Например, факт обслуживания той или иной модели автомобиля на отдельных станциях.

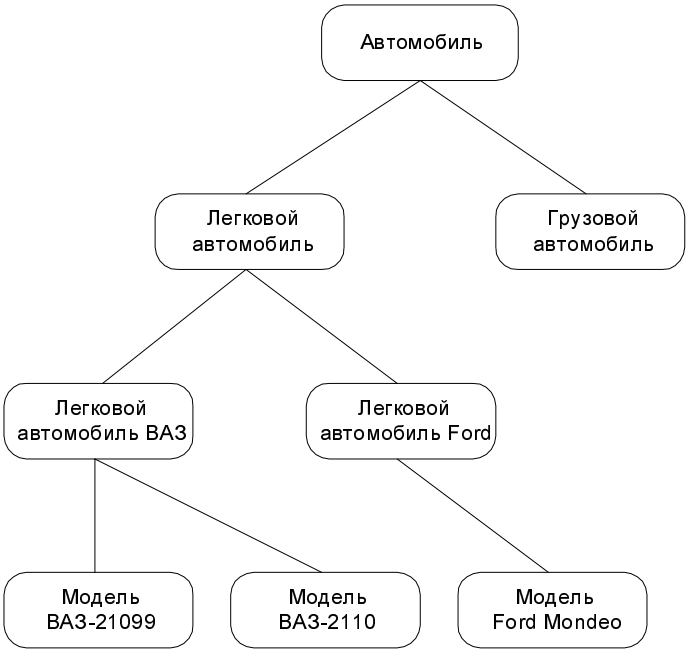


Рис. 2.7. Фрагмент семантической сети для представления иерархии классов "Автомобиль"

Построение моделей сложных систем, отражающих десятки различных типов объектов и связей между ними, привело, в конце 80-х годов прошлого столетия, к появлению большого числа различных графических нотаций, которые в той или иной степени были ориентированы на решение специальных классов задач. Сложилась парадоксальная ситуация, которая получила название "войны методов". Многие подходы, хотя и имели общие истоки, совершенно игнорировали другие альтернативные способы представления семантической информации. Наибольшее распространение в эти годы получил подход к моделированию программных систем, который получил название системный структурный анализ (ССА). Поскольку многие идеи ССА оказали непосредственное влияние на развитие языка UML, а используемая графическая нотация была реализована в некоторых CASE-средствах, далее приводится краткая характеристика основных компонентов данного направления графического моделирования программных систем.

2.2. Диаграммы структурного системного анализа

Под структурным системным анализом принято понимать метод исследования системы, который начинается с наиболее общего ее описания, с последующей детализацией представления отдельных аспектов ее поведения

и функционирования. При этом общая модель системы строится в виде некоторой иерархической структуры, которая отражает различные уровни абстракции с ограниченным числом компонентов на каждом из уровней. Одним из главных принципов структурного системного анализа является выделение на каждом из уровней абстракции только наиболее существенных компонентов или элементов системы.

В рамках данного направления программной инженерии принято рассматривать три графических нотации, получивших названия диаграмм: *диаграммы "сущность-связь"* (Entity-Relationship Diagrams, ERD), *диаграммы функционального моделирования* (Structured Analysis and Design Technique, SADT) и *диаграммы потоков данных* (Data Flow Diagrams, DFD).

2.2.1. Диаграммы "сущность-связь"

Данная нотация была предложена П. Ченом (P. Chen) в его известной работе 1976 года и получила дальнейшее развитие в работах Р. Баркера (R. Barker). Диаграммы "сущность-связь" (ERD) предназначены для графического представления моделей данных разрабатываемой программной системы и предлагают некоторый набор стандартных обозначений для определения данных и отношений между ними. С помощью этого вида диаграмм можно описать отдельные компоненты концептуальной модели данных и совокупность взаимосвязей между ними, имеющих важное значение для разрабатываемой системы.

Основными понятиями данной нотации являются понятия сущности и связи. При этом под *сущностью* (entity) понимается произвольное множество реальных или абстрактных объектов, каждый из которых обладает одинаковыми свойствами и характеристиками. В этом случае каждый рассматриваемый объект может являться *экземпляром* одной и только одной сущности, должен иметь уникальное имя или идентификатор, а также отличаться от других экземпляров данной сущности.

Примерами сущностей могут быть: банк, клиент банка, счет клиента, аэропорт, пассажир, рейс, компьютер, терминал, автомобиль, водитель. Каждая из сущностей может рассматриваться с различной степенью детализации и на различном уровне абстракции, что определяется конкретной постановкой задачи. Для графического представления сущностей используются специальные обозначения (рис. 2.8).

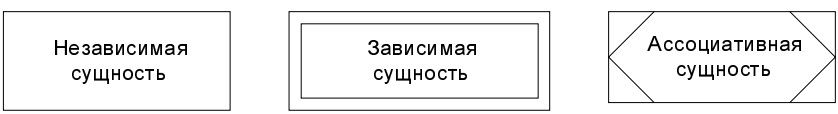


Рис. 2.8. Графические изображения для обозначения сущностей

Связь (relationship) определяется как отношение или некоторая ассоциация между отдельными сущностями. Примерами связей могут являться родст-

венные отношения типа "отец-сын" или производственные отношения типа "начальник-подчиненный". Другой тип связей задается отношениями "иметь в собственности" или "обладать свойством". Различные типы связей графически изображаются в форме ромба с соответствующим именем данной связи (рис. 2.9).

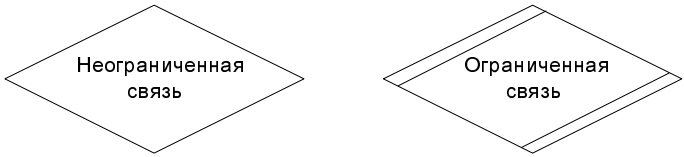


Рис. 2.9. Графические изображения для обозначения связей

Графическая модель данных строится таким образом, чтобы связи между отдельными сущностями отражали не только семантический характер соответствующего отношения, но и дополнительные аспекты обязательности связей, а также кратность участвующих в данных отношениях экземпляров сущностей.

Примечание

Как нетрудно заметить, имеется определенное сходство между понятиями сущность и объект, однако в языке UML понятие сущности имеет дополнительное содержание, которое будет рассмотрено далее в *главе 5*. Что касается понятия связь, то здесь мы имеем один из примеров некорректного перевода английского термина relationship на русский язык, выполненного, возможно, лицами, далекими от строгого использования математической нотации. Действительно, для термина *связь* в английском используется специальный термин – link, который в нотации языка UML имеет семантику экземпляра ассоциации. Но...об этом в свое время.

Рассмотрим в качестве простого примера ситуацию, которая описывается двумя сущностями: "сотрудник" и "компания". При этом в качестве связи естественно использовать отношение принадлежности сотрудника к данной компании. Если принять во внимание дополнительные условия, такие, например, как в компании работают несколько сотрудников, и эти сотрудники не могут быть работниками других компаний, то данная информация может быть представлена графически в виде следующей диаграммы "сущность-связь" (рис. 2.10). На данном рисунке буква "N" около связи означает тот факт, что в компании могут работать более одного сотрудника, при этом значение N заранее не фиксируется. Цифра "1" на другом конце связи означает, что сотрудник может работать только в одной конкретной компании, т. е. не допускается прием на работу сотрудников по совместительству из других компаний или учреждений.

Рис. 2.10. Диаграмма "сущность-связь" для примера сотрудников некоторой компании

Несколько иная ситуация складывается в случае рассмотрения сущностей "сотрудник" и "проект" и связи "участвует в работе над проектом" (рис. 2.11). Поскольку в общем случае один сотрудник может участвовать в разработке нескольких проектов, а в разработке одного проекта могут принимать участие несколько сотрудников, то данная связь является многозначной. Данный факт специально отражается на диаграмме указанием букв "N" и "M" около соответствующих сущностей, при этом выбор конкретных букв не является принципиальным.

Рис. 2.11. Диаграмма "сущность-связь" для примера сотрудников, участвующих в работе над проектами

Рассмотренные две диаграммы могут быть объединены в одну, на которой будет представлена информация о сотрудниках компании, участвующих в разработке проектов данной компании (рис. 2.12). При этом может быть введена дополнительная связь, характеризующая проекты данной компании.

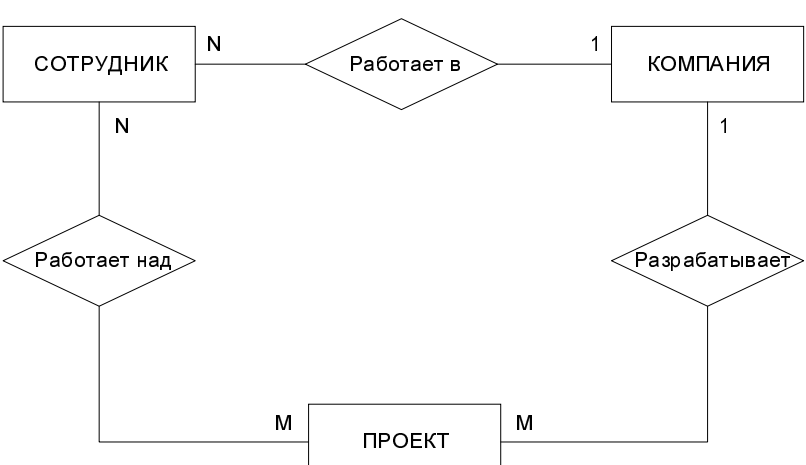


Рис. 2.12. Диаграмма "сущность-связь" для общего примера компании

Примечание

На указанных диаграммах могут быть отражены более сложные зависимости между отдельными сущностями, которые отражают обязательность выполнения некоторых дополнительных условий, определяемых спецификой решаемой задачи и моделируемой предметной области. В частности, могут быть отражены связи подчинения одной сущности другой или введены ограничения на действие отдельных связей. В подобных случаях используются дополнительные графические обозначения, отражающие особенности соответствующей семантики (см. рис. 2.8, 2.9).

Ограниченность диаграмм ERD проявляется при конкретизации концептуальной модели в более детальное представление моделируемой программной системы, которое кроме статических связей должно содержать информацию о поведении или функционировании отдельных ее компонентов. Для этих целей в рамках ССА используется другой тип диаграмм, получивших название диаграмм потоков данных. А сейчас перейдем к диаграммам SADT.

2.2.2. Диаграммы функционального моделирования

Начало разработки диаграмм функционального моделирования относится к середине 1960-х годов, когда Дуглас Т. Росс предложил специальную технику моделирования, получившую название SADT (Structured Analysis & Design Technique). Военно-воздушные силы США использовали методику SADT в качестве части своей программы *интеграции компьютерных и промышленных технологий* (Integrated Computer Aided Manufacturing, ICAM) и назвали ее IDEF (Icam DEFinition). Целью программы ICAM было увеличение эффективности компьютерных технологий в сфере проектирования новых средств вооружений и ведения боевых действий. Одним из результатов этих исследований являлся вывод о том, что описательные языки не эффективны для документирования и моделирования процессов функционирования сложных систем. Подобные описания на естественном языке не обеспечивают требуемого уровня непротиворечивости и полноты, имеющих доминирующее значение при решении задач моделирования.

В рамках программы ICAM было разработано несколько графических языков моделирования, которые получили следующие названия.

- Нотация IDEF0 — для документирования процессов производства и отображения информации об использовании ресурсов на каждом из этапов проектирования систем.
- Нотация IDEF1 — для документирования информации о производственном окружении систем.
- Нотация IDEF2 — для документирования поведения системы во времени.
- Нотация IDEF3 — специально для моделирования бизнес-процессов.

Примечание

Нотация IDEF2 никогда не была полностью реализована. Нотация IDEF1 в 1985 году была расширена и переименована в IDEF1X. Методология IDEF-SADT нашла применение в правительственных и коммерческих организациях, поскольку на тот период вполне удовлетворяла различным требованиям, предъявляемым к моделированию широкого класса систем.

В начале 1990 года специально образованная группа пользователей IDEF (IDEF Users Group), в сотрудничестве с Национальным институтом по стандартизации и технологии США (National Institutes for Standards and Technology, NIST), предприняла попытку создания стандарта для IDEF0 и IDEF1X. Эта попытка оказалась успешной и завершилась принятием в 1993 году стандарта правительства США, известного как FIPS для этих двух технологий IDEF0 и IDEF1X. В течение последующих лет этот стандарт продолжал активно развиваться и послужил основой для реализации в некоторых CASE-средствах, наиболее известным из которых является AllFusion Process Modeler (новое название BPwin) фирмы Computer Associates.

Методология IDEF-SADT представляет собой совокупность методов, правил и процедур, предназначенных для построения функциональной модели системы какой-либо предметной области. Функциональная модель SADT отображает структуру процессов функционирования системы и ее отдельных подсистем, т. е. выполняемые ими действия и связи между этими действиями. Для этой цели строятся специальные модели, которые позволяют в наглядной форме представить последовательность определенных действий. Исходными строительными блоками любой модели IDEF0 процесса являются *деятельность* (activity) и *стрелки* (arrows).

Рассмотрим кратко эти основные понятия методологии IDEF-SADT, которые используются при построении диаграмм функционального моделирования. *Деятельность* представляет собой некоторое действие или набор действий, которые имеют фиксированную цель и приводят к некоторому конечному результату. Иногда деятельность в нотации IDEF0 называют процессом (или что хуже — функцией). Модели IDEF0 отслеживают различные виды деятельности системы, их описание и взаимодействие с другими процессами. На диаграммах деятельность или процесс изображается прямоугольником, который называется блоком. *Стрелка* служит для обозначения некоторого носителя или воздействия, которые обеспечивают перенос данных или объектов от одной деятельности к другой. Стрелки также необходимы для описания того, что именно производит деятельность и какие ресурсы она потребляет. Это так называемые роли стрелок — ICOM — сокращение первых букв от названий соответствующих стрелок IDEF0. При этом различают стрелки четырех видов:

- I (Input) — вход, т. е. все, что поступает в процесс или потребляется процессом.

- C (Control) — управление или ограничения на выполнение операций процесса.
- O (Output) — выход или результат процесса.
- M (Mechanism) — механизм, который используется для выполнения процесса.

Методология IDEF0 однозначно определяет, каким образом изображаются на диаграммах стрелки каждого вида ICOM. Стрелка Вход (Input) выходит из левой стороны рамки рабочего поля и входит слева в прямоугольник процесса. Стрелка Управление (Control) входит и выходит сверху. Стрелка Выход (Output) выходит из правой стороны процесса и входит в правую сторону рамки. Стрелка Механизм (Mechanism) входит в прямоугольник процесса снизу. Таким образом, базовое представление процесса на диаграммах IDEF0 имеет следующий вид (рис. 2.13).

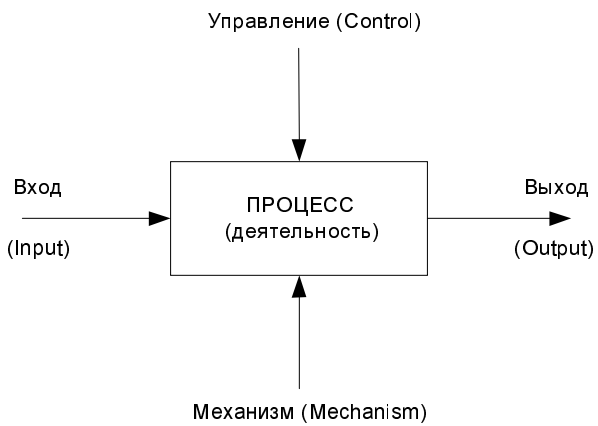


Рис. 2.13. Обозначение процесса и стрелок ICOM на диаграммах IDEF0

Техника построения диаграмм представляет собой главную особенность методологии IDEF-SADT. Место соединения стрелки с блоком определяет тип интерфейса. При этом все функции моделируемой системы и интерфейсы на диаграммах представляются в виде соответствующих блоков процессов и стрелок ICOM. Управляющая информация входит в блок сверху, в то время как информация, которая подвергается обработке, изображается с левой стороны блока. Результаты процесса представляются как выходы процесса и показываются с правой стороны блока. В качестве механизма может выступать человек или автоматизированная система, которые выполняют данную операцию. Соответствующий механизм на диаграмме представляется стрелкой, которая входит в блок процесса снизу.

Одной из наиболее важных особенностей методологии IDEF-SADT является постепенное введение все более детальных представлений модели системы

по мере разработки отдельных диаграмм. Построение модели IDEF-SADT начинается с представления всей системы в виде простейшей диаграммы, состоящей из одного блока процесса и стрелок ICOM, служащих для изображения основных видов взаимодействия с объектами вне системы. Поскольку исходный процесс представляет всю систему как единое целое, данное представление является наиболее общим и подлежит дальнейшей декомпозиции.

Для иллюстрации основных идей методологии IDEF-SADT рассмотрим следующий простой пример. В качестве процесса будем представлять деятельность по оформлению кредита в банке. Входом данного процесса является заявка от клиента на получение кредита, а выходом — соответствующий результат, т. е. непосредственно кредит. При этом управляющими факторами являются правила оформления кредита, которые регламентируют условия получения соответствующих финансовых средств в кредит. Механизмом данного процесса является служащий банка, который уполномочен выполнить все операции по оформлению кредита. Пример исходного представления процесса оформления кредита в банке изображен на рис. 2.14.

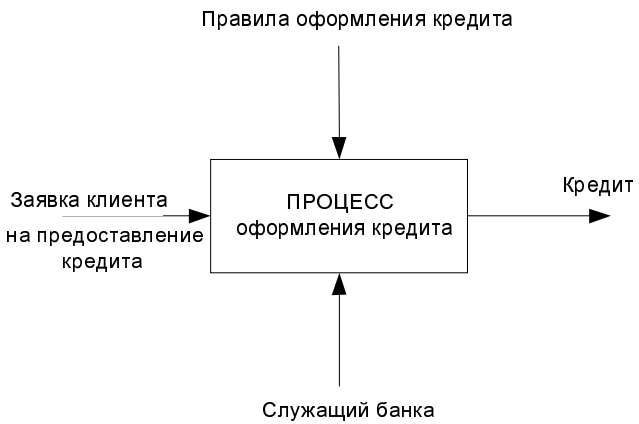


Рис. 2.14. Пример исходной диаграммы IDEF-SADT для процесса оформления кредита в банке

В конечном итоге модель IDEF-SADT представляет собой серию иерархически взаимосвязанных диаграмм с сопроводительной документацией, которая разбивает исходное представление сложной системы на отдельные составные части. Детали каждого из основных процессов представляются в виде более детальных процессов на других диаграммах. В этом случае каждая диаграмма нижнего уровня является декомпозицией некоторого процесса из более общей диаграммы. Поэтому на каждом шаге декомпозиции более общая диаграмма конкретизируется на ряд более детальных диаграмм.

В настоящее время диаграммы структурного системного анализа IDEF-SADT продолжают использоваться целым рядом организаций для построения и детального анализа функциональной модели существующих на предприятии бизнес-процессов, а также для реинжиниринга и разработки новых бизнес-процессов. Основной недостаток данной методологии связан с отсутствием явных средств для объектно-ориентированного представления моделей сложных систем. Хотя некоторые аналитики отмечают важность знания и применения нотации IDEF-SADT, ограниченные возможности этой методологии применительно к реализации соответствующих графических моделей в объектно-ориентированном программном коде существенно сужают диапазон решаемых с ее помощью задач.

2.2.3. Диаграммы потоков данных

Основой данной методологии графического моделирования информационных систем является специальная технология построения диаграмм потоков данных DFD (Data Flow Diagram). В разработке методологии DFD приняли участие многие аналитики, среди которых следует отметить Э. Йордона (E. Yourdon). Он является автором одной из первых графических нотаций DFD. В настоящее время наиболее распространенной является так называемая нотация Гейна-Сарсона (Gene-Sarson), основные элементы которой будут рассмотрены в этом разделе.

Модель системы в контексте DFD представляется в виде некоторой информационной модели, основными компонентами которой являются различные потоки данных, которые переносят информацию от одной подсистемы к другой. Каждая из подсистем выполняет определенные преобразования входного потока данных и передает результаты обработки информации в виде потоков данных для других подсистем.

Основными компонентами диаграмм потоков данных являются:

- внешние сущности
- процессы
- системы/подсистемы
- накопители данных или хранилища
- потоки данных

Внешняя сущность представляет собой материальный объект или физическое лицо, которые могут выступать в качестве источника или получателя информации. Определение некоторого объекта или системы в качестве внешней сущности не является строго фиксированным. Хотя внешняя сущность находится за пределами границ рассматриваемой системы, в процессе дальнейшего анализа некоторые внешние сущности могут быть перенесены внутрь диаграммы модели системы. С другой стороны, отдельные процессы

могут быть вынесены за пределы диаграммы и представлены как внешние сущности. Примерами внешних сущностей могут служить: клиенты организации, заказчики, персонал, поставщики.

Внешняя сущность обозначается прямоугольником с тенью (рис. 2.15), внутри которого указывается ее имя. При этом в качестве имени рекомендуется использовать существительное в именительном падеже. Иногда внешнюю сущность называют также *терминатором*.

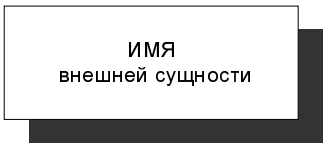


Рис. 2.15. Изображение внешней сущности на диаграмме потоков данных

Процесс представляет собой совокупность операций по преобразованию входных потоков данных в выходные, в соответствии с определенным алгоритмом или правилом. Хотя физически процесс может быть реализован различными способами, наиболее часто подразумевается программная реализация процесса. Процесс, на диаграмме потоков данных, изображается прямоугольником с закругленными вершинами (рис. 2.16), разделенным на три секции или поля горизонтальными линиями. Поле номера процесса служит для идентификации последнего. В среднем поле указывается имя процесса. В качестве имени рекомендовано использовать глагол в неопределенной форме с необходимыми дополнениями. Нижнее поле содержит указание на способ физической реализации процесса.

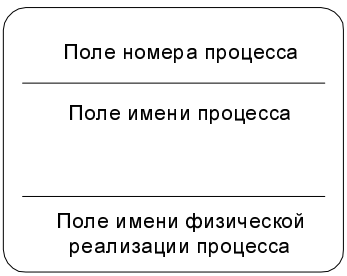


Рис. 2.16. Изображение процесса на диаграмме потоков данных

Информационная модель системы строится как некоторая иерархическая схема в виде так называемой контекстной диаграммы, на которой исходная модель последовательно представляется в виде модели подсистем соответствующих процессов преобразования данных. При этом *подсистема* или система на контекстной диаграмме DFD изображается так же, как и процесс — прямоугольником с закругленными вершинами (рис. 2.17).

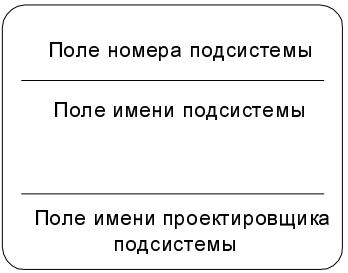


Рис. 2.17. Изображение подсистемы на диаграмме потоков данных

Накопитель данных или хранилище представляет собой абстрактное устройство или способ хранения информации, перемещаемой между процессами. Предполагается, что данные можно в любой момент поместить в накопитель и через некоторое время извлечь, причем физические способы помещения и извлечения данных могут быть произвольными. Накопитель данных может быть физически реализован различными способами, но наиболее часто предполагается его реализация в электронном виде на магнитных носителях. Накопитель данных на диаграмме потоков данных изображается прямоугольником с двумя полями (рис. 2.18). Первое поле служит для указания номера или идентификатора накопителя, который начинается с буквы "D". Второе поле служит для указания имени. При этом в качестве имени накопителя рекомендуется использовать существительное, которое характеризует способ хранения соответствующей информации.

Рис. 2.18. Изображение накопителя на диаграмме потоков данных

Наконец, *поток данных* определяет качественный характер информации, передаваемой через некоторое соединение от источника к приемнику. Реальный поток данных может передаваться по сети между двумя компьютерами или любым другим способом, допускающим извлечение данных и их восстановление в требуемом формате. Поток данных на диаграмме DFD изображается линией со стрелкой на одном из ее концов, при этом стрелка показывает направление потока данных. Каждый поток данных имеет свое собственное имя, отражающее его содержание.

Таким образом, информационная модель системы в нотации DFD строится в виде диаграмм потоков данных, которые графически представляются

с использованием соответствующей системы обозначений. В качестве примера рассмотрим упрощенную модель процесса получения некоторой суммы наличными по кредитной карточке клиентом банка. Внешними сущностями данного примера являются клиент банка и, возможно, служащий банка, который контролирует процесс обслуживания клиентов. Накопителем данных может быть база данных о состоянии счетов отдельных клиентов банка. Отдельные потоки данных отражают характер передаваемой информации, необходимой для обслуживания клиента банка. Соответствующая модель для данного примера может быть представлена в виде диаграммы потоков данных (рис. 2.19).

Рис. 2.19. Пример диаграммы DFD для процесса получения некоторой суммы наличными по кредитной карточке

В настоящее время диаграммы потоков данных используются в некоторых CASE-средствах для построения информационных моделей систем обработки данных. Основной недостаток данной методологии также связан с отсутствием явных средств для объектно-ориентированного представления моделей сложных систем, а также для представления сложных алгоритмов обработки данных. Поскольку на диаграммах DFD не указываются характеристики времени выполнения отдельных процессов и передачи данных между процессами, то модели систем, реализующих синхронную обработку данных, не могут быть адекватно представлены в нотации DFD. Все эти особенности методологии структурного системного анализа ограничили возможности ее широкого применения и послужили основой для включения соответствующих средств в унифицированный язык моделирования.

2.3. Основные этапы развития UML

Отдельные языки объектно-ориентированного моделирования стали появляться в период между серединой 1970-х и концом 1980-х годов, когда различные исследователи и программисты предлагали свои подходы к ООАП. В период между 1989—1994 гг. общее число наиболее известных языков моделирования возросло с 10 до более чем 50. Многие пользователи испытывали серьезные затруднения при выборе языка ООАП, поскольку ни один из них не удовлетворял всем требованиям, предъявляемым к построению моделей сложных систем. Принятие отдельных методик и графических нотаций в качестве стандартов (IDEF0, IDEF1X) не смогло изменить сложившуюся ситуацию непримиримой конкуренции между ними в начале 90-х годов, которая тоже получила название "войны методов".

К середине 1990-х некоторые из методов были существенно улучшены и приобрели самостоятельное значение при решении различных задач ООАП. Наиболее известными в этот период становятся:

- метод Гради Буча (Grady Booch), получивший условное название Booch или Booch'91, Booch Lite (позже — Booch'93);
- метод Джеймса Румбаха (James Rumbaugh), получивший название Object Modeling Technique — OMT (позже — OMT-2);
- метод Айвара Джекобсона (Ivar Jacobson), получивший название Object-Oriented Software Engineering — OOSE.

Каждый из этих методов был ориентирован на поддержку отдельных этапов ООАП. Например, метод OOSE содержал средства представления вариантов использования, которые имеют существенное значение на этапе анализа требований в процессе проектирования бизнес-приложений. Метод OMT-2 наиболее подходил для анализа процессов обработки данных в информационных системах. Метод Booch'93 нашел наибольшее применение на этапах проектирования и разработки различных программных систем.

История развития языка UML берет начало с октября 1994 года, когда Гради Буч и Джеймс Румбах из Rational Software Corporation начали работу по унификации методов Booch и OMT. Хотя сами по себе эти методы были достаточно популярны, совместная работа была направлена на изучение всех известных объектно-ориентированных методов с целью объединения их достоинств. При этом Г. Буч и Дж. Румбах сосредоточили усилия на полной унификации результатов своей работы. Проект так называемого *унифицированного метода* (Unified Method) версии 0.8 был подготовлен и опубликован в октябре 1995 года. Осенью того же года к ним присоединился А. Джекобсон, главный технолог из компании Objectory AB (Швеция), с целью интеграции своего метода OOSE с двумя предыдущими.

Вначале авторы методов Booch, OMT и OOSE предполагали разработать унифицированный язык моделирования только для этих трех методик.

С одной стороны, каждый из этих методов был проверен на практике и показал свою конструктивность при решении отдельных задач ООАП. Это давало основание для дальнейшей их модификации на основе устранения имеющегося несоответствия отдельных понятий и обозначений. С другой стороны, унификация семантики и нотации должна была обеспечить некоторую стабильность на рынке объектно-ориентированных CASE-средств, которая необходима для успешного продвижения соответствующих программных инструментариев. Наконец, совместная работа давала надежду на существенное улучшение всех трех методов.

Начиная работу по унификации своих методов, Г. Буч, Дж. Румбах и А. Джекобсон сформулировали следующие требования к языку моделирования. Он должен:

- позволять моделировать не только программное обеспечение, но и более широкие классы систем и бизнес-приложений, с использованием объектно-ориентированных понятий;
- явным образом обеспечивать взаимосвязь между базовыми понятиями для моделей концептуального и физического уровней;
- обеспечивать масштабируемость моделей, что является важной особенностью сложных многоцелевых систем;
- быть понятен аналитикам и программистам, а также должен поддерживаться специальными инструментальными средствами, реализованными на различных компьютерных платформах.

Разработка системы обозначений или нотации для ООАП оказалась непохожей на разработку нового языка программирования. Во-первых, необходимо было решить две проблемы.

1. Должна ли данная нотация включать в себя спецификацию требований?
2. Следует ли расширять данную нотацию до уровня языка визуального программирования?

Во-вторых, было необходимо найти удачный баланс между выразительностью и простотой языка. С одной стороны, слишком простая нотация ограничивает круг потенциальных проблем, которые могут быть решены с помощью соответствующей системы обозначений. С другой стороны, слишком сложная нотация создает дополнительные трудности для ее изучения и применения аналитиками и программистами. В случае унификации существующих методов необходимо учитывать интересы специалистов, которые уже имеют опыт работы с ними, поскольку внесение серьезных изменений в новую нотацию может повлечь за собой непонимание или неприятие ее пользователями прежних методик. Чтобы исключить неявное сопротивление со стороны отдельных специалистов, необходимо учитывать интересы самого широкого круга пользователей. Последующая работа над языком UML должна была учесть все эти обстоятельства.

В этот период поддержка разработки языка UML становится одной из целей консорциума OMG (Object Management Group). Хотя консорциум OMG был образован еще в 1989 году с целью разработки предложений по стандартизации объектных и компонентных технологий CORBA, язык UML приобрел статус второго стратегического направления в работе OMG. Именно в рамках OMG создается команда разработчиков под руководством Ричарда Соли, которая будет обеспечивать дальнейшую работу по унификации и стандартизации языка UML. В июне 1995 года OMG организовала совещание всех крупных специалистов и представителей входящих в консорциум компаний по методологиям ООАП, на котором впервые в международном масштабе была признана целесообразность поиска индустриальных стандартов в области языков моделирования под эгидой OMG.

Усилия Г. Буча, Дж. Румбаха и А. Джекобсона привели к появлению первых документов, содержащих описание собственно языка UML версии 0.9 (июнь 1996 г.) и версии 0.91 (октябрь 1996 г.). Имевшие статус запроса предложений RTP (Request For Proposals), эти документы послужили своеобразным катализатором для широкого обсуждения языка UML различными категориями специалистов. Первые отзывы и реакция на язык UML указывали на необходимость его дополнения отдельными понятиями и конструкциями.

В это же время стало ясно, что некоторые компании и организации видят в языке UML линию стратегических интересов для своего бизнеса. Компания Rational Software вместе с несколькими организациями, изъявившими желание выделить ресурсы для разработки строгого определения версии 1.0 языка UML, учредила консорциум партнеров UML, в который первоначально вошли такие компании, как Digital Equipment Corp., HP, i-Logix, Intellicorp, IBM, ICON Computing, MCI Systemhouse, Microsoft, Oracle, Rational Software, TI и Unisys. Эти компании обеспечили поддержку последующей работы по более точному и строгому определению нотации, что привело к появлению версии 1.0 языка UML. В январе 1997 года был опубликован документ с описанием языка UML 1.0 как начальный вариант ответа на запрос предложений RTP. Эта версия языка моделирования была достаточно хорошо определена, обеспечивала требуемую выразительность и мощность и предполагала решение широкого класса задач.

Примечание

Всеми партнерами консорциума осознается важность стандартизации языка UML, которая является необходимой основой для последующей разработки инструментальных CASE-средств. При этом разработка каждого стандарта в рамках консорциума OMG начинается с выпуска документа, называемого *запросом предложений* (Request For Proposals, RFP). Документы RFP в соответствии с установленной в OMG процедурой явно формулируют цели предстоящей разработки, требования, которым должны удовлетворять предложения, претендующие на стандартизацию, и объявляются сроки их представления. Отдельные RFP являются официальными документами, которыми руководствуются разработчики вариантов проектов соответствующих стандартов.

Инструментальные CASE-средства и диапазон их практического применения в большой степени зависят от удачного определения семантики и нотации соответствующего языка моделирования. Специфика языка UML заключается в том, что он определяет семантическую метамодель, а не модель конкретного интерфейса и способы представления или реализации компонентов.

Из более чем 800 компаний и организаций, входящих в настоящее время в состав консорциума OMG, особую роль продолжает играть Rational Software Corporation, которая стояла у истоков разработки языка UML. Эта компания разработала и выпустила в продажу одно из первых инструментальных CASE-средств Rational Rose 98, в котором была реализована нотация различных диаграмм языка UML.

Примечание

В феврале 2003 г. компания Rational Software Corporation была приобретена всемирно известной IBM, и с этого момента она имеет официальное название — IBM Rational Software. Соответствующим образом добавилась приставка ко всей линейке ее программных продуктов, включая последнюю версию IBM Rational Rose 2003. Тем не менее в данном историческом очерке используется ее прежнее название, под которым эта компания известна всем корпоративным программистам и аналитикам.

В январе 1997 года целый ряд других компаний, среди которых были IBM, ObjecTime, Platinum Technology и некоторые другие, представили на рассмотрение OMG свои собственные ответы на запрос предложений RFP. В дальнейшем эти компании присоединились к партнерам UML, предлагая включить в язык UML некоторые свои идеи. В результате совместной работы с партнерами UML была предложена пересмотренная версия 1.1 языка UML. Основное внимание при разработке языка UML 1.1 было уделено достижению большей ясности семантики языка по сравнению с UML 1.0, а также учету предложений новых партнеров. Эта версия языка была представлена на рассмотрение OMG, одобрена и принята в качестве стандарта OMG в ноябре 1997 года.

В настоящее время все вопросы дальнейшей разработки языка UML сконцентрированы в рамках консорциума OMG. Соответствующая группа специалистов обеспечивает публикацию материалов, содержащих описание последующих версий языка UML и запросов предложений RFP по его стандартизации. Очередной этап развития данного языка закончился в марте 1999 года, когда консорциумом OMG было опубликовано описание языка UML 1.3 (alpha R5). Следующей версией языка UML стала версия 1.4, специфицированная в сентябре 2001 г., после выхода которой началась работа над следующей версией, которая была анонсирована как UML 2.0. Видимо, именно с этого момента стал проявляться интерес к языку UML со стороны прессы, которая, публикуя интервью с "Three amigos", неоднократно указывала на важные нововведения в этой перспективной версии.

Однако сроки принятия и публикации спецификации на язык UML версии 2.0, начиная с 2001 г., в силу различных причин постоянно откладывались. Возможно, для многих совершенной неожиданностью оказалось появление, в марте 2003 г., документации на версию UML 1.5, которая описана в соответствующем документе 03.03.01 — *"OMG Unified Modeling Language Specification"*. Именно эта версия является последней версией языка UML на момент написания книги и рассматривается в ней. История разработки и последующего развития языка UML графически представлена на рис. 2.20.

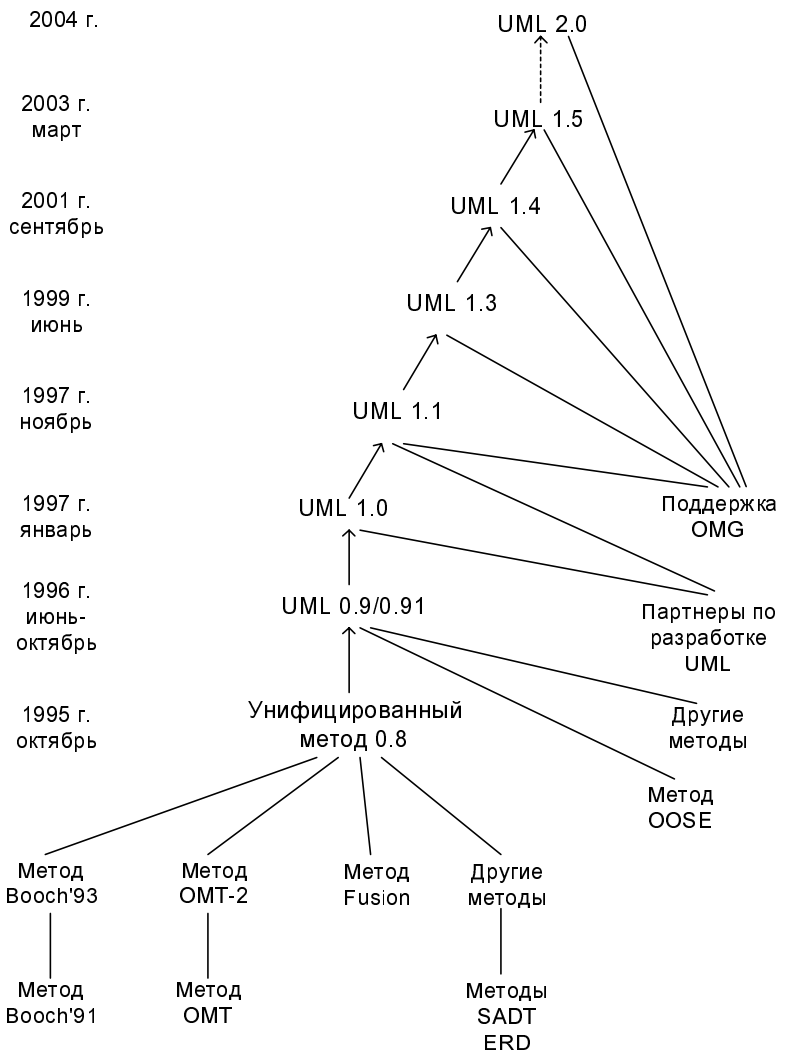


Рис. 2.20. История развития языка UML

Примечание

В настоящее время в Интернете, на сайте **www.omg.org**, на рассмотрение представлен проект спецификации на UML 2.0. Окончание обсуждения намечено на сентябрь 2003 г., а принятие стандарта — на апрель 2004 г. Однако как реально будут развиваться события — покажет время.

Статус языка UML определен как открытый для всех предложений по его доработке и совершенствованию. Сам язык UML не является чьей-либо собственностью и не запатентован кем-либо, хотя указанный выше документ защищен законом об авторском праве. В то же время аббревиатура UML, как и некоторые другие (OMG, CORBA, ORB), является торговой маркой их законных владельцев, о чем следует упомянуть в данном контексте.

Язык UML ориентирован для применения в качестве языка моделирования различными пользователями и научными сообществами для решения широкого класса задач ООАП. Многие специалисты по методологии, организации и поставщики инструментальных средств обязались использовать язык в своих разработках. При этом термин "унифицированный" в названии UML не является случайным и имеет два аспекта. С одной стороны, он фактически устраняет многие из несущественных различий между известными ранее языками моделирования и методиками построения диаграмм. С другой стороны, создает предпосылки для унификации различных моделей и этапов их разработки для широкого класса систем, не только программного обеспечения, но и бизнес-процессов. Семантика языка UML определена таким образом, что она не является препятствием для последующих усовершенствований при появлении новых концепций моделирования.

В этой связи следует отметить внимание гиганта компьютерной индустрии — компании Microsoft, к технологии UML. Еще в октябре 1996 г. Microsoft и Rational Software Corporation объявили о своем стратегическом партнерстве, которое, по их общему мнению, способно оказать решающее влияние на рынок средств объектно-ориентированной разработки программ. При этом Microsoft лицензировала у Rational Software некоторые технологии визуального моделирования, в результате чего был разработан Microsoft Visual Modeler for Visual Basic. В свою очередь Rational Software лицензировала у Microsoft Visual Basic и Microsoft Repository, разрабатываемые вместе с Texas Instruments. При создании языка UML Microsoft внесла свой вклад в интеграцию UML со своими стандартами типа ActiveX и COM и в использование языка UML со своей технологией Microsoft Repository.

На основе технологии UML компании Microsoft, Rational Software и другие поставщики средств разработки программных систем разработали единую информационную модель, которая получила название UML Information Model. Предполагается, что эта модель даст возможность различным программам, поддерживающим идеологию UML, обмениваться между собой компонентами и описаниями. Все это позволит создать стандартный интер-

фейс между средствами разработки приложений и средствами визуального моделирования.

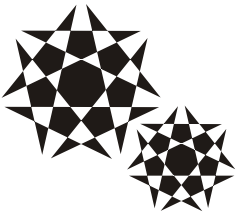
В настоящее время на рынке CASE-средств представлено несколько десятков программных инструментов, поддерживающих нотацию языка UML и обеспечивающих интеграцию, включая прямую и обратную генерацию кода программ, с наиболее распространенными языками и средами программирования, такими как MS Visual C++, Java, Object Pascal/Delphi, Power Builder, MS Visual Basic, Forte, Ada, Smalltalk. Поскольку при разработке языка UML были приняты во внимание многие передовые идеи и методы, можно ожидать, что на очередные версии языка UML также окажут влияние и другие перспективные технологии и концепции. Кроме того, на основе языка UML могут быть определены многие новые перспективные методы, поскольку сам язык UML может быть расширен без переопределения его исходных конструкций.

Примечание

Весьма важным событием стало появление в популярном RAD-средстве Borland Delphi 7 инструмента ModelMaker, предназначенного для работы с диаграммами языка UML с последующей генерацией кода на Delphi Pascal. В этой программной разработке нидерландской компании ModelMaker Tools реализована не только поддержка графической нотации языка UML, но и использование так называемых *паттернов* проектирования, которые еще год назад вызывали в лучшем случае улыбку, а в худшем – скепсис собеседника .

Подводя итог анализу методологии ООАП, истории появления и развития языка UML, можно утверждать следующее. Имеются все основания предполагать, что в ближайшие годы интерес к языку UML со стороны самых различных специалистов будет только возрастать. Язык UML не только станет основой для разработки и реализации во многих перспективных инструментальных RAD-средствах, но и в CASE-средствах визуального и имитационного моделирования. Более того, заложенные в языке UML потенциальные возможности могут быть использованы как для объектно-ориентированного моделирования систем, так и для документирования бизнес-процессов, а в более широком контексте — для представления знаний в интеллектуальных системах, которыми, по существу, станут перспективные сложные программно-технологические комплексы.

Глава 3



Основные компоненты языка UML

UML представляет собой общецелевой язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес-процессов и других систем. Он является простым и мощным средством моделирования, который может быть эффективно использован для построения концептуальных, логических и графических моделей сложных систем самого различного целевого назначения. Этот язык вобрал в себя наилучшие качества и опыт методов программной инженерии, которые с успехом использовались на протяжении последних лет при моделировании больших и сложных систем.

Язык UML основан на некотором числе базовых понятий, которые могут быть изучены и применены большинством программистов и разработчиков, знакомых с методами объектно-ориентированного анализа и проектирования. При этом базовые понятия могут комбинироваться и расширяться таким образом, что специалисты объектного моделирования получают возможность самостоятельно разрабатывать модели больших и сложных систем в самых различных областях приложений.

Конструктивное использование языка UML основывается на понимании общих принципов моделирования сложных систем и особенностей процесса объектно-ориентированного анализа и проектирования в частности. Выбор выразительных средств для построения моделей сложных систем предопределяет те задачи, которые могут быть решены с использованием данных моделей. При этом одним из основных принципов построения моделей сложных систем является принцип *абстрагирования*, который предписывает включать в модель только те аспекты проектируемой системы, которые имеют непосредственное отношение к выполнению системой своих функций или своего целевого предназначения. При этом все второстепенные детали опускаются, чтобы чрезмерно не усложнять процесс анализа и исследования полученной модели.

Другим принципом построения моделей сложных систем является принцип *многомодельности*. Этот принцип представляет собой утверждение о том, что

никакая единственная модель не может с достаточной степенью адекватности описывать различные аспекты сложной системы. Феномен сложной системы как раз и состоит в том, что никакая ее единственная модель не является достаточной для адекватного выражения всех особенностей моделируемой системы.

Применительно к методологии ООАП это означает, что достаточно полная модель сложной системы представляет собой некоторое число взаимосвязанных *представлений* (views), каждое из которых адекватно отражает некоторый аспект поведения или структуры системы. При этом наиболее общими представлениями сложной системы принято считать статическое и динамическое представления, которые в свою очередь могут подразделяться на другие, более частные представления.

Еще одним принципом прикладного системного анализа является принцип *иерархического построения* моделей сложных систем. Этот принцип предписывает рассматривать процесс построения модели на разных уровнях абстрагирования или детализации в рамках фиксированных представлений. При этом исходная или первоначальная модель сложной системы имеет наиболее общее представление (*метапредставление*). Такая модель строится на начальном этапе проектирования и может не содержать многих деталей и аспектов моделируемой системы.

Рис. 3.1. Общая схема взаимосвязей моделей и представлений сложной системы в процессе объектно-ориентированного анализа и проектирования

Таким образом, процесс ООАП можно представить как поуровневый спуск от наиболее общих моделей и представлений концептуального уровня к более частным и детальным представлениям логического и физического уровня. При этом на каждом из этапов ООАП данные модели последовательно дополняются все большим количеством деталей, что позволяет им более адекватно отражать различные аспекты конкретной реализации сложной системы. Общая схема взаимосвязей моделей ООАП представлена на рис. 3.1.

Примечание

Название "физическая модель" в терминологии ООАП и языка UML отличается от общепринятой трактовки этого термина в общей классификации моделей систем. В последнем случае под физической моделью системы понимают некоторую материальную конструкцию, обладающую свойствами подобия с формой оригинала. Примерами таких моделей могут служить модели технических систем (самолетов, кораблей), архитектурных сооружений (зданий, микрорайонов). Что касается использования этого термина в ООАП и языке UML, то здесь физическая модель отражает компонентный состав проектируемой системы с точки зрения ее реализации на некоторой технической базе и вычислительных платформах конкретных производителей.

3.1. Назначение языка UML

Язык UML предназначен для решения ряда задач.

1. *Предоставить в распоряжение пользователей легко воспринимаемый и выразительный язык визуального моделирования, специально предназначенный для разработки и документирования моделей сложных систем самого различного целевого назначения.*

   - Речь идет о том, что важным фактором дальнейшего развития и повсеместного использования методологии ООАП является интуитивная ясность и понятность основных конструкций соответствующего языка моделирования. Язык UML включает в себя не только абстрактные конструкции, для представления метамоделей систем, но и целый ряд конкретных понятий, имеющих вполне определенную семантику. Это позволяет языку UML одновременно достичь не только универсальности представления моделей для самых различных приложений, но и возможности описания достаточно тонких деталей реализации этих моделей применительно к конкретным системам.
   - Практика системного моделирования показала, что абстрактного описания языка на некотором метауровне недостаточно для разработчиков, которые ставят своей целью реализацию проекта системы в конкретные сроки. В настоящее время имеет место некоторый концептуальный разрыв между общей методологией моделирования сложных систем и конкретными инструментальными средствами быстрой разработки при-

ложений. Именно этот разрыв по замыслу OMG и призван заполнить язык UML.

- Отсюда вытекает важное следствие — для адекватного понимания базовых конструкций языка UML важно не только владеть некоторыми навыками объектно-ориентированного программирования, но и хорошо представлять себе общую проблематику процесса разработки моделей систем. Именно интеграция этих представлений образует новую парадигму ООАП, практическим следствием и центральным стержнем которой является язык UML.

2. *Снабдить исходные понятия языка UML возможностью расширения и специализации для более точного представления моделей систем в конкретной предметной области.*

- Хотя язык UML является формальным языком спецификаций, формальность его описания отличается от синтаксиса как традиционных формально-логических языков, так и известных языков программирования. Разработчики из OMG предполагают, что язык UML как никакой другой может быть приспособлен для конкретных предметных областей. Это становится возможным по той причине, что в самом описании языка UML заложен механизм расширения базовых понятий, который является самостоятельным элементом и имеет собственное описание в форме правил расширения.
- В то же время разработчики из OMG считают крайне нежелательным переопределение базовых понятий языка. Это может привести к неоднозначной интерпретации их семантики и возможной путанице. Базовые понятия языка UML не следует изменять больше, чем это необходимо для их расширения. Все пользователи должны быть способны строить модели систем для большинства обычных приложений с использованием только базовых конструкций языка UML без применения механизма расширения. При этом новые понятия и нотации целесообразно применять только в тех ситуациях, когда имеющихся базовых понятий явно недостаточно для построения моделей системы.
- Язык UML допускает также специализацию базовых понятий. Речь идет о том, что в конкретных приложениях пользователи должны уметь дополнять имеющиеся базовые понятия новыми характеристиками или свойствами, которые не противоречат семантике этих понятий.

3. *Описание языка UML должно поддерживать такую спецификацию моделей, которая не зависит от конкретных языков программирования и инструментальных средств проектирования программных систем.*

- Речь идет о том, что конструкции языка UML не должны зависеть от особенностей их реализации в известных языках программирования. Другими словами, хотя отдельные понятия языка UML семантически связаны с последними, их жесткая интерпретация в форме конструкций

программирования не может быть признана корректной. Разработчики из OMG считают необходимым свойством языка UML его контекстно-программную независимость.

- С другой стороны, язык UML должен обладать потенциальной возможностью реализации своих конструкций на том или ином языке программирования. Конечно, в первую очередь имеются в виду языки, поддерживающие концепцию ООП, такие как C++, Java, Object Pascal. Именно это свойство языка UML делает его современным средством решения задач моделирования сложных систем. В то же время, предполагается, что для программной поддержки конструкций языка UML могут быть разработаны специальные инструментальные CASE-средства. Наличие последних имеет принципиальное значение для широкого распространения и использования языка UML.

4. *Описание языка UML должно включать в себя семантический базис для понимания общих особенностей ООАП.*

- Говоря об этой особенности, имеют в виду самодостаточность языка UML для понимания не только его базовых конструкций, но что не менее важно — понимания общих принципов ООАП. В этой связи необходимо отметить, что поскольку язык UML не является языком программирования, а служит средством для решения задач объектно-ориентированного моделирования систем, описание языка UML должно по возможности включать в себя все необходимые понятия для ООАП. Без этого свойства язык UML может оказаться бесполезным и невостребованным большинством пользователей, которые не знакомы с проблематикой ООАП сложных систем.
- С другой стороны, какие бы то ни было ссылки на дополнительные источники, содержащие важную для понимания языка UML информацию, по мнению разработчиков из OMG, должны быть исключены. Это позволит избежать неоднозначного толкования принципиальных особенностей процесса ООАП и их реализации в форме базовых конструкций языка UML.

5. *Поощрять развитие рынка программных инструментальных средств, поддерживающих концепции объектных технологий.*

- Более 800 ведущих производителей программных и аппаратных средств, усилия которых сосредоточены в рамках OMG, видят перспективы развития современных информационных технологий и основу своего коммерческого успеха в широком продвижении на рынок инструментальных средств, поддерживающих объектные технологии. Говоря же об объектных технологиях, разработчики из OMG имеют в виду, прежде всего, совокупность технологических решений CORBA и UML. С этой точки зрения языку UML отводится роль базового средства для описания и документирования различных объектных компонентов CORBA.

6. *Способствовать распространению объектных технологий и соответствующих понятий ООАП.*
   - Эта задача тесно связана с предыдущей и имеет с ней много общего. Если исключить из рассмотрения рекламные заявления разработчиков об исключительной гибкости и мощности языка UML, а попытаться составить объективную картину возможностей этого языка, то можно прийти к следующему заключению. Следует признать, что усилия достаточно большой группы разработчиков были направлены на интеграцию в рамках языка UML многих известных техник визуального моделирования, которые успешно зарекомендовали себя на практике *(см. главу 2)*. Хотя это привело к усложнению языка UML по сравнению с известными нотациями структурного системного анализа, платой за сложность являются действительно высокая гибкость и изобразительные возможности уже первых версий языка UML. В свою очередь, использование языка UML для решения всевозможных практических задач будет только способствовать его дальнейшему совершенствованию, а значит и дальнейшему развитию объектных технологий и практики ООАП.
7. *Интегрировать в себя новейшие и наилучшие достижения практики ООАП.*
   - Язык UML непрерывно совершенствуется разработчиками, и основой этой работы является его дальнейшая интеграция с современными технологиями моделирования. При этом различные методы системного моделирования получают свое прикладное осмысление в рамках ООАП. В последующем эти методы могут быть включены в состав языка UML в форме дополнительных базовых понятий, наиболее адекватно и полно отражающие наилучшие достижения практики ООАП.

Чтобы решить указанные выше задачи, за свою недолгую историю язык UML претерпел определенную эволюцию. В результате описание самого языка UML стало нетривиальным, поскольку семантика базовых понятий включает в себя целый ряд перекрестных связей с другими понятиями и конструкциями языка. В связи с этим так называемое линейное или последовательное рассмотрение основных конструкций языка UML стало практически невозможным, т. к. одни и те же понятия могут использоваться при построении различных диаграмм или представлений. В то же время каждое из представлений модели обладает собственными семантическими особенностями, которые накладывают отпечаток на семантику базовых понятий языка в целом.

Примечание

Отмечая сложность описания языка UML, следует отметить присущую всем формальным языкам сложность их строгого задания, которая вытекает из необходимости в той или иной степени использовать естественный язык для спецификации

базовых примитивов. В этом случае естественный язык выступает в роли *метаязыка*, т. е. языка для описания формального языка. Поскольку естественный язык не является формальным, то и его применение для описания формальных языков в той или иной степени страдает неточностью. Хотя в задачи языка UML не входит анализ соответствующих логико-лингвистических деталей, однако эти особенности отразились на структуре описания языка UML, в частности, делая стиль описания всех его основных понятий полуформальным.

Учитывая эти особенности, принятая в книге последовательность изучения языка UML основывается на рассмотрении основных графических средств моделирования, а именно — канонических диаграмм. Все необходимые для построения диаграмм понятия вводятся по мере необходимости. Тем не менее, в этой главе следует остановиться на общих особенностях языка UML, которые в той или иной степени влияют на понимание его базовых конструкций.

3.2. Общая структура языка UML

С самой общей точки зрения описание языка UML состоит из двух взаимодействующих частей:

- семантика языка UML. Представляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке UML;
- нотация языка UML. Представляет собой графическую нотацию для визуального представления семантики языка UML.

Абстрактный синтаксис и семантика языка UML описываются с использованием некоторого подмножества нотации UML. В дополнение к этому, нотация UML описывает соответствие, или отображение графической нотации в базовые понятия семантики. Таким образом, с функциональной точки зрения эти две части дополняют друг друга. При этом семантика языка UML описывается на основе некоторой метамодели, имеющей три отдельных представления: абстрактный синтаксис, правила корректного построения выражений и семантику. Рассмотрение семантики языка UML предполагает некоторый "полуформальный" стиль изложения, который объединяет естественный и формальный языки для представления базовых понятий и правил их расширения.

Семантика определяется для двух видов объектных моделей: структурных моделей и моделей поведения. *Структурные* модели, известные также как статические модели, описывают структуру сущностей или компонентов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения. Модели *поведения*, называемые иногда динамическими моделями, описывают поведение или функционирование объектов системы, включая их методы, взаимодействие и сотрудничество между ними, а также процесс изменения состояний отдельных компонентов и системы в целом.

Для решения столь широкого диапазона задач моделирования разработана достаточно полная семантика для всех компонентов графической нотации. Требования семантики языка UML конкретизируются при построении отдельных видов диаграмм, последовательное рассмотрение которых служит темой второй части книги. Нотация языка UML включает в себя описание отдельных семантических элементов, которые могут применяться при построении диаграмм.

Формальное описание самого языка UML основывается на некоторой общей иерархической структуре модельных представлений, состоящей из четырех уровней:

- мета-метамодель;
- метамодель;
- модель;
- объекты пользователя.

Уровень *мета-метамодели* образует исходную формально-логическую основу для всех метамодельных представлений. Главное предназначение этого уровня состоит в том, чтобы определить язык для спецификации метамодели. Мета-метамодель определяет модель языка UML на самом высоком уровне абстракции и является наиболее компактным ее описанием. С другой стороны, мета-метамодель может специфицировать несколько метамоделей, чем достигается потенциальная гибкость включения дополнительных понятий. Хотя в книге этот уровень не рассматривается, он наиболее тесно связан с теорией формальных языков. Примерами понятий этого уровня служат метакласс, метаатрибут, метаоперация.

Примечание

Следует отметить, что семантика мета-метамодели не входит в описание языка UML. С одной стороны, это делает язык UML более простым для изучения, поскольку не требует знания общей теории формальных языков и формальной логики. С другой стороны, наличие мета-метамодели придает языку UML черты формально-логического языка, которые необходимы ему для того, чтобы обладать свойством непротиворечивости. Если эти особенности могут представляться мало интересными для многих программистов, то разработчики инструментальных средств никак не могут их игнорировать.

Метамодель является экземпляром или конкретизацией мета-метамодели. Главная задача этого уровня — определить язык для спецификации моделей. Этот уровень является более конструктивным, чем предыдущий, поскольку обладает более развитой семантикой базовых понятий. Все основные понятия языка UML — это понятия уровня метамодели. Примеры таких понятий: класс, атрибут, операция, компонент, ассоциация и многие другие. Именно рассмотрению семантики и графической нотации понятий уровня метамодели посвящена данная книга.

Модель в контексте языка UML является экземпляром метамодели в том смысле, что любая конкретная модель системы должна использовать только понятия метамодели, конкретизировав их применительно к данной ситуации. Это уровень для описания информации о конкретной предметной области. Однако если для построения модели используются понятия языка UML, то необходима полная согласованность понятий уровня модели с базовыми понятиями языка UML уровня метамодели. Примерами понятий уровня модели могут служить, например, имена полей проектируемой базы данных, такие как: имя и фамилия сотрудника, возраст, должность, адрес, телефон. При этом данные понятия используются лишь как имена соответствующих информационных атрибутов.

Конкретизация понятий модели происходит на уровне *объектов пользователя*. При этом совокупность объектов пользователя рассматривается как отдельный экземпляр модели, поскольку содержит конкретную информацию относительно того, чему в действительности соответствуют те или иные понятия модели. Примером объекта может служить следующая запись в проектируемой базе данных: "Илья Петров, 30 лет, иллюзионист, ул. Невидимая, 10-20, 100-0000".

Описание семантики языка UML предполагает рассмотрение базовых понятий только уровня *метамодели*, который представляет собой пример или частный случай уровня мета-метамодели. Метамодель UML является по своей сути скорее логической моделью, чем физической или моделью реализации. Особенность логической модели заключается в том, что она концентрирует внимание на декларативной или концептуальной семантике, опуская детали конкретной физической реализации моделей. При этом отдельные реализации, использующие данную логическую метамодель, должны быть согласованы с ее семантикой, а также поддерживать возможности импорта и экспорта отдельных логических моделей.

В то же время, логическая метамодель может быть реализована различными способами для обеспечения требуемого уровня производительности и надежности соответствующих инструментальных средств. В этом заключается недостаток логической модели, которая не содержит на уровне семантики требований, обязательных для ее эффективной последующей реализации. Однако согласованность метамодели с конкретными моделями реализации является обязательной для всех разработчиков программных средств, обеспечивающих поддержку языка UML.

Метамодель языка UML имеет довольно сложную структуру, которая включает в себя около 90 метаклассов, более 100 метаассоциаций и 50 стереотипов, число которых возрастает с появлением новых версий языка. Чтобы справиться с этой сложностью языка UML, все его элементы организованы специальным образом в логически связанные подмножества, получившие название пакетов. Поэтому рассмотрение языка UML на метамодельном уровне заключается в описании трех его наиболее общих логических блоков

или *пакетов*: основные элементы, элементы поведения и управление моделями.

Примечание

Говоря о пакетах в контексте общего описания языка, мы, по сути дела, приступаем к рассмотрению графической нотации языка UML. Дело в том, что для описания языка UML используются средства самого языка, и одним из таких средств является *пакет*. В общем случае пакет служит для группировки элементов модели. При этом сами элементы модели, которыми могут быть произвольные сущности, отнесенные к одному пакету, выступают в роли единого целого. Пакеты, так же как и другие элементы модели, могут быть вложены в другие пакеты. Важной особенностью языка UML является тот факт, что все виды элементов модели UML могут быть организованы в пакеты.

3.3. Пакеты в языке UML

Пакет — основной способ организации элементов модели в языке UML. Каждый пакет владеет всеми своими элементами, т. е. теми элементами, которые включены в него. Про соответствующие элементы пакета говорят, что они принадлежат пакету или входят в него. При этом каждый элемент может принадлежать только *одному* пакету. В свою очередь, одни пакеты могут быть вложены в другие пакеты. В этом случае первые называются *подпакетами*, поскольку все элементы подпакета будут принадлежать более общему пакету. Тем самым для элементов модели задается отношение вложенности пакетов, которое представляет собой иерархию.

Примечание

При рассмотрении отношения "пакет-подпакет" наиболее естественно ассоциировать его с более общим отношением "множество-подмножество", которое было рассмотрено *в главе 2*. Действительно, поскольку пакет можно рассматривать в качестве частного случая множества, такая интерпретация помогает нам использовать и графические средства для представления соответствующих отношений между пакетами.

Из *главы 2* нам также известно, что для графического представления иерархий могут использоваться графы специального вида, которые называются деревьями (см. рис. 2.5, 2.6). Однако в языке UML эти графические обозначения настолько модифицированы, что соответствующие ассоциации с общетеоретическими понятиями могут представлять определенную трудность для начинающих разработчиков. Тем не менее, на протяжении всей книги подчеркивается важность умения ассоциировать специальные конструкции языка UML с соответствующими понятиями теории множеств и системного моделирования, что, в некотором смысле, формирует стиль мышления системого аналитика. В противном случае не исключены досадные ошибки

не только на начальном этапе концептуализации предметной области, но и в процессе построения различных представлений систем.

В языке UML для визуализации пакетов разработана специальная символика или графическая нотация, которой мы и будем пользоваться в дальнейшем. Именно с описания этой системы обозначений мы приступим к изучению основных элементов данного языка.

Для графического изображения пакетов на диаграммах применяется специальный графический символ — большой прямоугольник с небольшим прямоугольником, присоединенным к левой части верхней стороны первого (рис. 3.2). Можно сказать, что визуально символ пакета напоминает пиктограмму папки в популярном графическом интерфейсе. Внутри большого прямоугольника может записываться информация, относящаяся к данному пакету. Если такой информации нет, то внутри большого прямоугольника записывается имя пакета, которое должно быть уникальным в пределах рассматриваемой модели (рис. 3.2, *а*). Если же такая информация имеется, то имя пакета записывается в верхнем маленьком прямоугольнике (рис. 3.2, *б*).

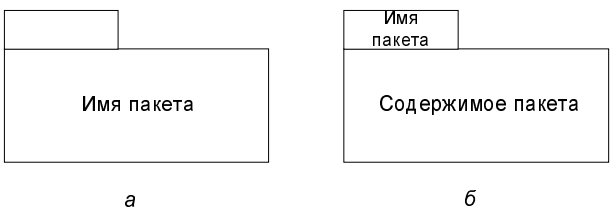


Рис. 3.2. Графическое изображение пакетов в языке UML

Примечание

Говоря об имени пакета, следует остановиться на общем соглашении об именах в языке UML. В данном случае именем пакета может быть строка текста, содержащее любое число букв, цифр и некоторых специальных знаков. С целью удобства спецификации пакетов принято в качестве их имен использовать существительные, возможно, с пояснительными словами, например, контроллер, графический интерфейс, форма ввода данных.

Перед именем пакета может помещаться строка текста, содержащая некоторое ключевое слово. Такими ключевыми словами являются заранее определенные в языке UML слова, которые получили название *стереотипов*. Стереотипами для пакетов являются слова facade, framework, stub и topLevel. В качестве содержимого пакета могут выступать имена его отдельных элементов и их свойства, такие как видимость элементов за пределами пакета. Более подробно стереотипы и видимость элементов будут рассмотрены в последующих главах книги.

Конечно, сами по себе пакеты могут найти ограниченное применение, поскольку содержат лишь информацию о входящих в их состав элементах модели. Не менее важно представить графически отношения, которые могут иметь место между отдельными пакетами. Как и в теории графов, для визуализации отношений в языке UML применяются отрезки линий, внешний вид которых имеет смысловое содержание.

Одним из типов отношений между пакетами является отношение вложенности или включения пакетов друг в друга. С одной стороны, в языке UML это отношение может быть изображено без использования линий простым размещением одного пакета-прямоугольника внутри другого пакета-прямоугольника (рис. 3.3). Так, в данном случае пакет с именем Пакет_1 содержит в себе два подпакета: Пакет_2 и Пакет_3.

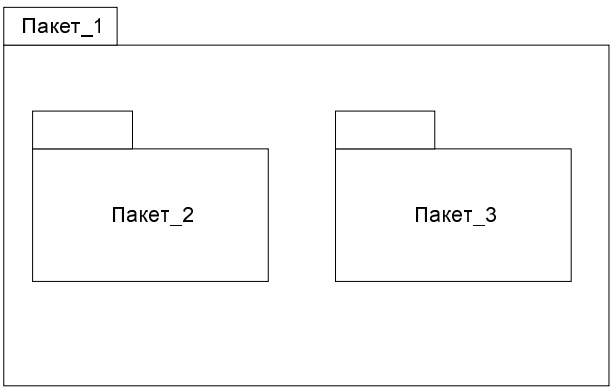


Рис. 3.3. Графическое изображение вложенности пакетов друг в друга

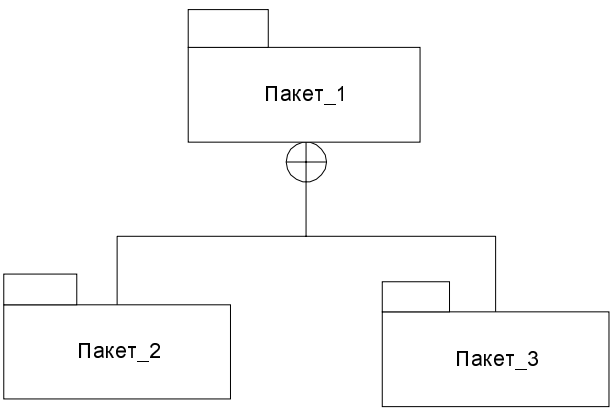


Рис. 3.4. Графическое изображение вложенности пакетов друг в друга с помощью явной визуализации отношения включения

С другой стороны, это же отношение может быть изображено с помощью отрезков линий аналогично графическому представлению дерева. В этом случае наиболее общий пакет (метапакет или контейнер) изображается в верхней части рисунка, а его подпакеты — уровнем ниже. Метапакет соединяется с подпакетами сплошной линией, на конце которой, примыкающей к метапакету, изображается специальный символ — ⊕ (знак плюс в кружочке). Этот символ означает, что подпакеты являются "собственностью" или частью контейнера и, кроме этих подпакетов, контейнер не содержит никаких других подпакетов. Рассмотренный ранее пример (рис. 3.3) может быть представлен с помощью явной визуализации отношения включения (рис. 3.4).

На графических диаграммах между пакетами могут указываться и другие типы отношений, часть из которых будет рассмотрена в последующих главах книги.

3.4. Основные пакеты метамодели языка UML

Возвращаясь к рассмотрению языка UML, напомним, что основой его представления на метамодельном уровне является описание трех его логических блоков или пакетов: Основные элементы, Элементы поведения и Управление моделями (рис. 3.5).

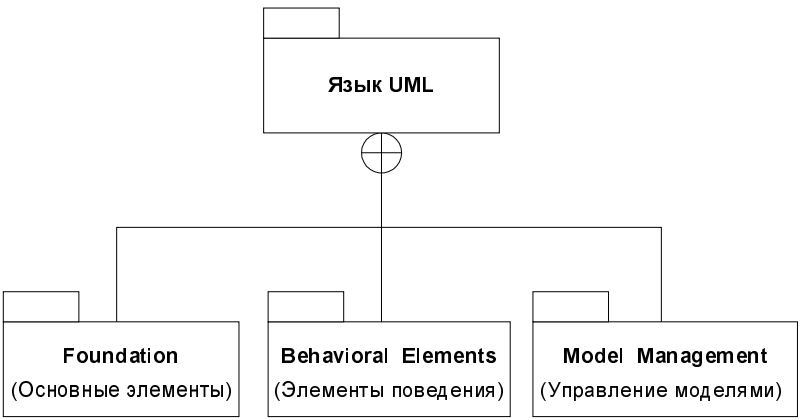


Рис. 3.5. Основные пакеты метамодели языка UML

Эти пакеты в свою очередь также включают в себя другие подпакеты. Например, пакет Основные элементы состоит из подпакетов: Элементы ядра, Механизмы расширения и Типы данных (рис. 3.6). При этом пакет Элементы ядра описывает базовые понятия и принципы включения в структуру метамодели основных понятий языка, таких как метаклассы, метаассоциации и метаатрибуты. Пакет Механизмы расширения задает правила

расширения семантики и графической нотации базовых элементов моделей. Пакет Типы данных определяет основные структуры данных, используемых в языке UML.

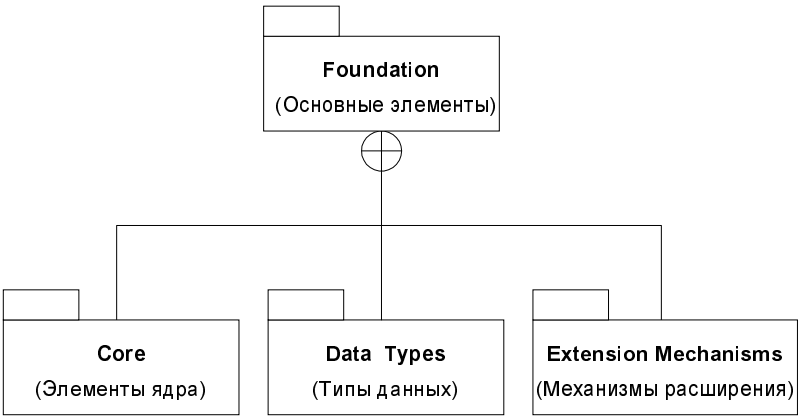


Рис. 3.6. Подпакеты пакета Основные элементы языка UML

3.4.1. Пакет Основные элементы

Ниже дается краткая характеристика элементов каждого из перечисленных подпакетов, входящих в состав пакета Основные элементы. Более полное рассмотрение отдельных компонентов метамодели будет представлено в главах, посвященных изучению отдельных видов канонических диаграмм. Последние аккумулируют в себе не только различные представления моделируемой системы, но и более детально раскрывают семантические особенности применения базовых конструкций языка UML в процессе построения конкретных моделей.

Пакет Элементы ядра

Пакет Элементы ядра является наиболее фундаментальным из всех подпакетов, которые входят в пакет Основные элементы языка UML. Он определяет основные абстрактные и конкретные компоненты, необходимые для разработки объектных моделей. При этом абстрактные компоненты метамодели не имеют экземпляров или примеров и используются исключительно для уточнения других компонентов модели. Конкретные компоненты метамодели имеют экземпляры и отражают особенности представления лиц, которые разрабатывают объектные модели.

Примечание

Следует отметить присущую развитым языкам представления знаний в целом и языку UML в частности избыточность выразительных средств. Речь идет

о том, что одна и та же моделируемая сущность или система может быть представлена средствами языка UML по-разному. При этом разные разработчики могут построить объектные модели одной и той же системы, существенно отличающиеся не только формой своего представления, но и составом используемых в модели компонентов.

Пакет Элементы ядра специфицирует базовые конструкции, требуемые для описания исходной метамодели, и определяет архитектурный "скелет" для присоединения дополнительных конструкций языка, таких как метаклассы, метаассоциации и метаатрибуты. Хотя пакет Элементы ядра содержит семантику, достаточную для определения всей оставшейся части языка UML, он не является мета-метамоделью UML.

В этот пакет входят основные метаклассы языка UML: класс (Class), атрибут (Attribute), ассоциация (Association), ассоциация-класс (AssociationClass), конец ассоциации (AssociationEnd), свойство поведения (BehavioralFeature), классификатор (Classifier), ограничение (Constraint), тип данных (DataType), зависимость (Dependency), элемент (Element), право на элемент (ElementOwnership), свойство (Feature), обобщение (Generalization), элемент отношения обобщения (GeneralizableElement), интерфейс (Interface), метод (Method), элемент модели (ModelElement), пространство имен (Namespace), операция (Operation), параметр (Parameter), структурное свойство (StructuralFeature), правила правильного построения выражений (Well-formedness rules).

Пакет Механизмы расширения

Пакет Механизмы расширения специфицирует порядок включения в модель элементов с уточненной семантикой, а также модификацию отдельных компонентов языка UML для более точного отражения специфики моделируемых систем. Механизм расширения определяет семантику для стереотипов, ограничений и помеченных значений. Хотя язык UML обладает богатым множеством понятий и нотаций для моделирования типичных программных систем, реально разработчик может столкнуться с необходимостью включить в модель дополнительные свойства или нотации, которые не определены явно в языке UML. При этом разработчики часто сталкиваются с необходимостью включения в модель графической информации, такой, например, как дополнительные значки и украшения.

Для этой цели в языке UML предусмотрены три механизма расширения, которые могут использоваться совместно или раздельно для определения новых элементов модели с отличающимися семантикой, нотацией и свойствами от специфицированных в метамодели языка UML элементов. Такими механизмами являются: ограничение (Constraint), стереотип (Stereotype) и помеченное значение (TaggedValue).

Таким образом, механизмы расширения языка UML предназначены для выполнения следующих задач:

- уточнения или специализации достаточно общих модельных элементов при разработке конкретных моделей на языке UML;
- определения таких расширений языка UML, которые зависят от специфики моделируемого процесса или от языка реализации программного кода;
- присоединения произвольной семантической или несемантической информации к элементам модели.

Хотя вопросы расширения метамодели UML выходят за пределы настоящей книги, следует знать о потенциальной возможности явного добавления в язык UML новых метаклассов и других метаконструкций. При этом, однако, необходимо соблюдать правило порождения новых метаклассов от уже имеющихся в языке UML. Эта возможность единственно зависит от свойств отдельных инструментальных средств, поддерживающих язык UML, или от особенностей мета-метамодельного представления самого процесса ООАП.

Наиболее важные из встроенных механизмов расширения основываются на понятии *стереотип*. Стереотипы обеспечивают некоторый способ классификации модельных элементов на уровне объектной модели и возможность добавления в язык UML "виртуальных" метаклассов с новыми атрибутами и семантикой. Другие встроенные механизмы расширения основываются на понятии *списка свойств*, содержащего помеченные значения и ограничения. Эти механизмы обеспечивают пользователю возможность включения дополнительных свойств и семантики непосредственно в отдельные элементы модели.

Пакет Типы данных

Пакет Типы данных специфицирует различные типы данных, которые могут использоваться в языке UML. Этот пакет имеет более простую по сравнению с другими пакетами внутреннюю структуру и описание, поскольку предполагается, что семантика соответствующих понятий хорошо известна.

В метамодели UML типы данных используются для объявления типов атрибутов классов. Они записываются в форме строк текста на диаграммах и не имеют отдельного значка "тип данных". Благодаря этому происходит уменьшение размеров диаграмм без потери информации. Однако каждая из одинаковых записей для некоторого типа данных должна соответствовать одному и тому же типу данных в модели. При этом типы данных, используемые в описании языка UML, могут отличаться от типов данных, которые определяет разработчик для своей модели на языке UML. Типы данных в последнем случае будут являться частным случаем или экземплярами метакласса *типы данных*, который определен в метамодели.

При задании типа данных наиболее часто применяется неформальная конструкция, которая получила называние *перечисления*. Речь идет о множестве допустимых значений атрибута, которое наделяется некоторым отношением

порядка. При этом упорядоченность значений либо указывается явно заданием первого и последнего элементов списка, либо следует неявно в случае простого типа данных, как, например, для множества натуральных чисел. В пакете Типы данных определены способы спецификации перечислений для корректного задания допустимых значений атрибутов.

Для определения различных типов данных в языке UML используются как простые конструкции: целое число (Integer), строка (String), имя (Name), булевский (Boolean), кратность (Multiplicity), тип видимости (VisibilityKind), диапазон кратности (MultiplicityRange), так и более сложные: выражение (Expression), булевское выражение (BooleanExpression), тип агрегирования (AggregationKind), тип изменения (ChangeableKind), геометрия (Geometry), отображение (Mapping), выражение-процедура (ProcedureExpression), тип псевдосостояния (PseudostateKind), выражение времени (TimeExpression).

3.4.2. Пакет Элементы поведения

Этот пакет является самостоятельной компонентой языка UML и, как следует из его названия, специфицирует динамику поведения в нотации UML. Пакет Элементы поведения состоит из подпакетов: Общее поведение, Кооперации, Варианты использования, Конечные автоматы, Графы деятельности и Действия (рис. 3.7). Ниже дается краткая характеристика каждого из этих подпакетов.

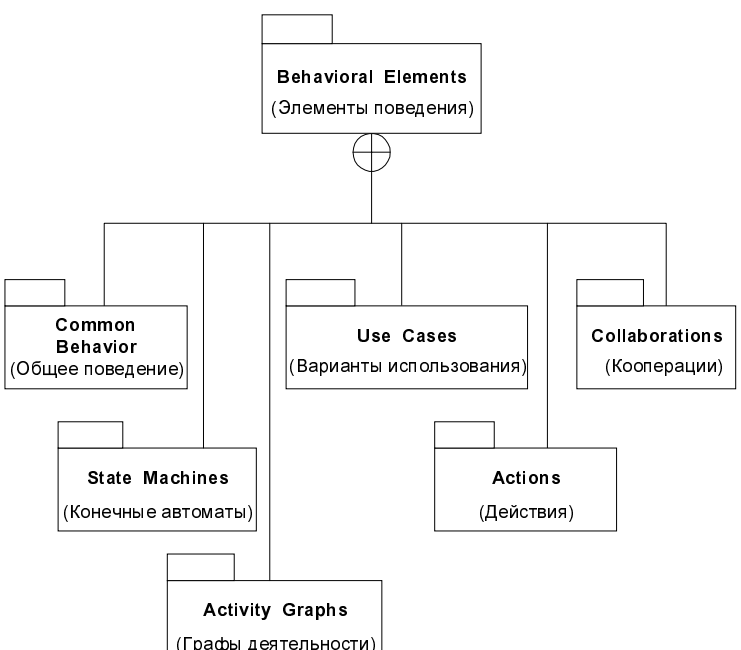


Рис. 3.7. Подпакеты пакета Элементы поведения языка UML

Пакет Общее поведение

Пакет Общее поведение является наиболее фундаментальным из всех подпакетов и определяет базовые понятия ядра, необходимые для всех элементов поведения. В этом пакете специфицирована семантика для динамических элементов, которые включены в другие подпакеты элементов поведения. В пакет Общее поведение входит достаточно большое число элементов, таких как: экземпляр (Instance), объект (Object), исключение (Exception), связь (Link), сигнал (Signal), значение данных (DataValue), процедура (Procedure), связь атрибутов (AttributeLink) и т. д.

Наиболее важным понятием пакета Общее поведение является *объект*. Под объектом в языке UML понимается отдельный экземпляр или пример класса, структура и поведение которого полностью определяется порождающим этот объект классом. Предполагается, что все без исключения объекты, порожденные одним и тем же классом, имеют совершенно одинаковую структуру и поведение, хотя каждый из этих объектов может иметь свое собственное множество связей атрибутов. При этом каждая связь атрибута относится к некоторому экземпляру, обычно к значению данных. Это множество может быть модифицировано согласно спецификации отдельного атрибута в описании класса.

Рассматривая данный пакет, следует отметить, что в языке UML под поведением понимается не только процесс изменения атрибутов объектов в результате выполнения операций над их значениями, но и такие процедуры, как создание и уничтожение самих объектов. При этом динамика взаимодействия объектов, которая определяет их поведение, описывается с помощью специальных понятий, таких как *сигналы* и *действия*.

Пакет Кооперации

Пакет Кооперации специфицирует контекст поведения при использовании элементов модели для выполнения отдельной задачи. В нем задается семантика понятий, которые необходимы для ответа на вопрос "Как различные элементы модели взаимодействуют между собой с точки зрения структуры?". Этот пакет использует конструкции, определенные в пакетах Основные элементы языка UML и Общее поведение.

В частности, в пакет Кооперации входят элементы: кооперация (Collaboration), взаимодействие (Interaction), сообщение (Message), роль ассоциации (AssociationRole), роль классификатора (ClassifierRole), роль конца ассоциации (AssociationEndRole). Как можно догадаться из названия пакета, его элементы непосредственно используются при построении диаграмм кооперации. Понятие кооперации имеет важное значение для представления взаимодействия элементов модели с точки зрения классификаторов и ассоциаций. Особенности их применения будут более детально рассмотрены при изучении диаграммы кооперации *(см. главу 6)*.

Пакет Варианты использования

Пакет Варианты использования специфицирует поведение при включении в модель специальных конструкций, которые в языке UML называются *актерами* и *вариантами использования*. Эти понятия служат для определения функциональности моделируемой сущности, такой как система. Особенность элементов этого пакета состоит в том, что они используются для первоначального определения поведения сущности без спецификации ее внутренней структуры.

Примечание

Объединение в языке UML средств концептуализации исходных требований к проектируемой системе и структуризации ее внутренних компонентов с достаточно богатой семантикой применяемых для этого элементов имеет важное значение для построения адекватных моделей сложных систем. Действительно, ограниченность традиционных моделей состоит в том, что они не позволяют одновременно описывать статические или структурные свойства системы и динамику ее поведения. Попытки совместного решения данных проблем сталкиваются с отсутствием единой символики для обозначения близких по смыслу системных понятий. Язык UML удачно выделяет базовые понятия, которые необходимы при построении таких моделей. Более того, если этих понятий окажется недостаточно для разработки какого-то конкретного проекта, то сам разработчик может расширить базовые понятия и даже включить в модель собственные конструкции, согласованные с метамоделью языка UML.

В пакет Варианты использования кроме уже упомянутых элементов актер (Actor) и вариант использования (UseCase) входят: расширение (Extend), точка расширения (ExtensionPoint), включение (Include) и экземпляр варианта использования (UseCaseInstance). Более подробно некоторые из этих понятий будут рассмотрены при описании диаграмм вариантов использования *(см. главу 4)*.

Пакет Конечные автоматы

Пакет Конечные автоматы специфицирует поведение при построении моделей с использованием систем переходов для конечного множества состояний. В нем определено множесто понятий, которые необходимы для представления поведения модели в виде дискретного пространства с конечным числом состояний и переходов.

Формализм конечного автомата, который используется в языке UML, отличается от формализма теории автоматов своей объектной ориентацией. Конечные автоматы являются основным средством моделирования поведения различных элементов языка UML. Они, например, могут использоваться для моделирования поведения индивидуальных сущностей, таких как экземпляры классов или объектов, а также для спецификации взаимодействий между сущностями, такими как кооперации. Формализм конечных автоматов

дополнительно обеспечивает семантический базис для графов деятельности, которые являются частным случаем конечного автомата.

В пакет Конечные автоматы входят элементы: состояние (State), переход (Transition), событие (Event), конечный автомат (StateMachine), простое состояние (SimpleState), составное состояние (CompositeState), псевдосостояние (PseudoState), конечное состояние (FinalState), сторожевое условие (Guard) и некоторые другие.

Как уже отмечалось *(см. главу 2)*, одним из ключевых понятий при моделировании динамических свойств систем является состояние. При этом под *состоянием* в языке UML понимается абстрактный метакласс, используемый для моделирования ситуации или процесса, в ходе которых имеет место (обычно неявное) выполнение некоторого инвариантного условия. Примером такого условия может быть состояние ожидания объектом выполнения некоторого внешнего события, например, запроса или передачи управления. С другой стороны, состояние может использоваться для моделирования динамических условий, таких как процесс выполнения некоторой деятельности. В этом случае момент начала выполнения деятельности является переходом объекта в соответствующее состояние.

Более подробно понятия этого пакета будут рассмотрены при изучении диаграмм состояний *(см. главу 8)*.

Пакет Графы деятельности

В этом пакете специфицированы понятия, которые могут быть использованы для построения моделей процессов с использованием нетриггерных переходов. В нем определено множество понятий, которые необходимы для представления логики протекания процессов и выполнения процедур или алгоритмов, включая бизнес-процессы и документооборот компаний и фирм.

В пакет Графы деятельности входят элементы: граф деятельности (ActivityGraph), разбиение (Partition), состояние действия (ActionState), состояние поддеятельности (SubactivityState), состояние вызова (CallState) и некоторые другие. Более подробно понятия этого пакета будут рассмотрены при изучении диаграмм деятельности *(см. главу 9)*.

Пакет Действия

Пакет Действия является новым пакетом языка UML версии 1.5 и специфицирует синтаксис и семантику выполняемых действий и процедур, включая семантику времени их выполнения. Этот пакет в свою очередь состоит из нескольких подпакетов, в которых определяются те или иные конкретные действия. Эти действия связаны с выполнением различных процедур, таких как создание и уничтожение объектов (CreateObjectAction и DestroyObjectAction), создание и уничтожение связей (CreateLinkAction и DestroyLinkAction), вычисление (Computation Actions), чтение и запись (Read Write Actions),

передача сообщений (Messaging Actions) и т. д. Детальный анализ этих действий выходит за пределы этой книги.

3.4.3. Пакет Управление моделями

Пакет Управление моделями (Model Management) специфицирует базовые элементы языка UML, которые необходимы для формирования всех модельных представлений. Именно в нем определяется семантика модели (Model), пакета (Package) и подсистемы (Subsystem). Эти элементы служат своеобразными контейнерами для группировки других элементов модели.

Пакет является метаклассом в языке UML и предназначен, как отмечалось ранее, для организации других элементов модели, таких как другие пакеты, классификаторы и ассоциации. Пакет может также содержать ограничения и зависимости между элементами модели в самом пакете. Предполагается, что каждый элемент пакета имеет видимость только внутри данного пакета. Это означает, что за пределами пакета никакой его элемент не может быть использован, если нет дополнительных указаний на импорт или доступ к отдельным элементам пакета. Со своей стороны, пакеты со всем своим содержимым определены в некотором пространстве имен, которое определяет единственность использования имен всех элементов модели. Другими словами, имя каждого элемента модели должно быть единственным в некотором пространстве имен, которое, являясь само элементом модели, может быть вложено в более общее пространство имен.

Модель является подклассом пакета и представляет собой абстракцию физической системы, которая предназначена для вполне определенной цели. Именно эта цель предопределяет те компоненты, которые должны быть включены в модель, и те, рассмотрение которых не является обязательным. Другими словами, модель отражает релевантные аспекты физической системы, оказывающие непосредственное влияние на достижение поставленной цели. В прикладных задачах цель обычно задается в форме исходных требований к системе, которые, в свою очередь, в языке UML записываются в виде вариантов использования системы.

В языке UML для одной и той же физической системы могут быть определены различные модели, каждая из которых специфицирует систему с различных точек зрения. Примерами таких моделей являются логическая модель, модель проектирования, модель вариантов использования и другие. При этом каждая такая модель имеет свою собственную точку зрения на физическую систему и свой собственный уровень абстракции. Модели, как и пакеты, могут быть вложенными друг в друга. Со своей стороны, пакет может включать в себя несколько различных моделей одной и той же системы, и в этом состоит один из важнейших механизмов разработки моделей на языке UML. В общем случае модель системы в контексте языка UML

включает в себя модель анализа и модель проектирования, что с использованием обозначений пакетов может быть изображено следующим образом (рис. 3.8).

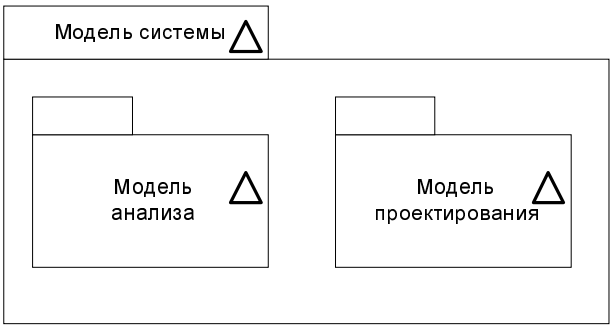


Рис. 3.8. Изображение модели системы в виде пакетов моделей анализа и проектирования

Подсистема есть просто группировка элементов модели, которые специфицируют некоторое простейшее поведение физической системы. В метамодели UML подсистема является подклассом как пакета, так и классификатора. Элементы подсистемы делятся на две части — спецификацию поведения и его реализацию.

Для графического представления подсистемы применяется специальное обозначение — прямоугольник, как в случае пакета, но дополнительно разделенный на три секции (рис. 3.9). При этом в верхнем маленьком прямоугольнике изображается символ, по своей форме напоминающий "вилку" и указывающий на подсистему. Имя подсистемы вместе с необязательными ключевым словом или стереотипом записывается внутри большого прямоугольника. Однако при наличии строк текста внутри большого прямоугольника имя подсистемы может быть записано рядом с обозначением "вилки".

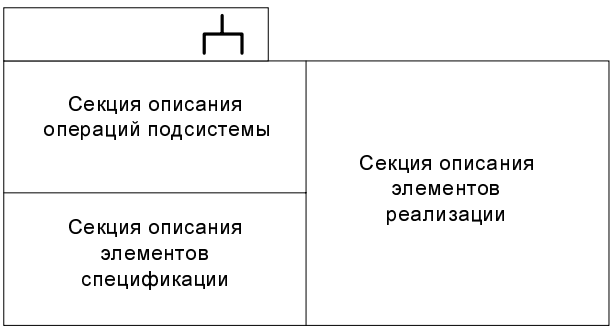


Рис. 3.9. Графическое изображение подсистемы в языке UML

Операции подсистемы записываются в левой верхней секции, ниже указываются элементы спецификации, а справа от вертикальной линии — элементы реализации. При этом два последних раздела помечаются соответствующими метками: "Элементы спецификации" и "Элементы реализации". Секция операций никак не помечается. Если в подсистеме отсутствуют секции, то они не отображаются на схеме.

3.5. Специфика описания метамодели языка UML

Метамодель языка UML описывается на некотором полуформальном языке с использованием трех видов представлений:

- абстрактный синтаксис;
- правила правильного построения выражений;
- семантика.

Абстрактный синтаксис представляет собой модель для описания некоторой части языка UML, предназначенной для построения диаграмм классов на основе описаний систем на естественном языке. Возможности абстрактного синтаксиса в языке UML довольно ограничены и имеют отношение только к интерпретации обозначений отдельных компонентов диаграмм, связей между компонентами и допустимых дополнительных обозначений. К элементам абстрактного синтаксиса относятся некоторые ключевые слова и значения отдельных атрибутов базовых понятий уровня метамодели, которые имеют фиксированное обозначение в виде текста на естественном языке.

Правила правильного построения выражений используются для задания дополнительных ограничений или свойств, которыми должны обладать те или иные компоненты модели. Поскольку исходным понятием ООП является понятие класса, его общими свойствами должны обладать все экземпляры, которые в этом смысле должны быть инвариантны друг другу. Для задания этих инвариантных свойств классов и отношений необходимо использовать специальные выражения некоторого формального языка, в рамках UML получившего название *языка объектных ограничений* (Object Constraint Language, OCL). Хотя язык OCL и использует естественный язык для формулировки правил правильного построения выражений, особенности его применения являются темой самостоятельного обсуждения. Основные особенности языка OCL рассмотрены в *Приложении*.

Семантика языка UML описывается в основном на естественном языке, но может включать в себя некоторые дополнительные обозначения, вытекающие из связей определяемых понятий с другими понятиями. Семантика понятий раскрывает их смысл или содержание. Сложность описания семантики языка UML заключается именно в метамодельном уровне представлений

его основных конструкций. С одной стороны, понятия языка UML имеют абстрактный характер (ассоциация, композиция, агрегация, сотрудничество, состояние). С другой стороны, каждое из этих понятий допускает свою конкретизацию на уровне модели (сотрудник, отдел, должность, стаж).

Сложность описания семантики языка UML вытекает из этой двойственности понятий. Здесь мы должны придерживаться традиционных правил изложения, поскольку понимание семантики носит индуктивный характер и требует для своей интерпретации примеров уровня модели и объекта. Иллюстрация абстрактных понятий на примере конкретных свойств и отношений, а также их значений позволяет акцентировать внимание на общих инвариантах этих понятий, что совершенно необходимо для понимания их семантики.

Примечание

Таким образом, метамодель языка UML может рассматриваться как комбинация графической нотации (специальных обозначений), некоторого формального языка и естественного языка. При этом следует иметь в виду, что существует некоторый теоретический предел, который ограничивает описание метамодели средствами самой метамодели. Именно в подобных случаях используется естественный язык, обладающий наиболее выразительными возможностями.

Хотя сам термин "естественный язык" далеко не однозначен и порождает целый ряд дополнительных вопросов, здесь мы ограничимся его трактовкой в форме обычного текста на русском и, возможно, английском языках. Как бы ни хотелось некоторым из отечественных разработчиков, полностью избежать использования английского при описании языка UML не удастся. Тем не менее, если исключить написание стандартных элементов и некоторых ключевых слов, то во всех остальных случаях под естественным языком можно понимать русский без специальных оговорок.

Для придания формального характера моделям UML использование естественного языка должно строго соответствовать определенным правилам. Например, описание семантики языка UML может включать в себя фразы типа "Сущность *А* обладает способностью" или "Сущность *Б* есть сущность *В*". В каждом из этих случаев мы будем понимать смысл фраз, руководствуясь традиционным пониманием предложений русского языка. Однако этого может оказаться недостаточно для более формального представления знаний о рассматриваемых сущностях. Тогда необходимо дополнительно специфицировать семантику этих простых фраз, для чего рекомендуется использовать следующие правила:

- явно указывать в тексте экземпляр некоторого метакласса. Речь идет о том, что в естественной речи мы часто опускаем слово "пример" или "экземпляр", говоря просто "класс". Так, фразу "Атрибут **возраст** класса **сотрудник** имеет значение **30 лет**" следует записать более точно, а именно: "Атрибут **возраст** экземпляра класса **сотрудник** имеет значение **30 лет**";

- в каждый момент используется только то значение слова, которое приписано имени соответствующей конструкции языка UML. Все дополнительные особенности семантики должны быть указаны явным образом, без каких бы то ни было неявных предположений;
- термины языка UML могут включать только один из допустимых префиксов, таких как под-, супер- или мета-. При этом сам термин с префиксом записывается одним словом.

В дополнение к этому будут использоваться следующие правила выделения текста:

- если используются ссылки на конструкции языка UML, а не на их представления в метамодели, следует применять обычный текст, без какого бы то ни было выделения;
- имена метаклассов являются элементом нотации языка UML и представляют собой существительное и, возможно, присоединенное к нему прилагательное. В этом случае имя метакласса на английском записывается одним словом с выделением каждой составной части имени заглавной буквой (например, ModelElement, StructuralFeature);
- имена метаассоциаций и ассоциаций классов записываются аналогичным образом (например, ElementReference);
- имена других элементов языка UML также записываются одним словом, но должны начинаться со строчной буквы (например, ownedElement, allContents);
- имена метаатрибутов, которые принимают булевы значения, всегда начинаются с префикса "is" (например, isAbstract);
- перечислимые типы должны всегда заканчиваться словом "Kind" (например, AggregationKind);
- при ссылках в тексте на метаклассы, метаассоциации, метаатрибуты и т. д. должны всегда использоваться в точности те их имена, которые указаны в модели;
- имена стандартных обозначений (стереотипов) заключаются в угловые кавычки и начинаются со строчной буквы (например, <<type>>).

Рассмотренные выше правила выделения текста имеют непосредственное отношение к англоязычным терминам языка UML. Поскольку вопросы локализации языка UML до настоящего времени не нашли своего отражения в работе OMG, отечественным специалистам придется самостоятельно дополнять эти правила на случай использования в качестве естественного русского языка. В книге мы будем придерживаться двух дополнительных рекомендаций.

- При описании семантики языка UML все имена его стандартных элементов (метаклассов, метаассоциаций, метаатрибутов) допускается записывать

на русском с дополнительным указанием оригинального имени на английском языке. При этом, хотя имена стандартных элементов могут состоять из нескольких слов, согласно сложившейся отечественной традиции, будем их записывать раздельно (например, класс ассоциации, элемент модели, пространство имен).

- При разработке конкретных моделей систем в форме диаграмм языка UML целесообразно применять оригинальные англоязычные термины, придерживаясь описанных ранее правил (кроме, возможно, пояснительного текста на русском). Причина этой рекомендации вполне очевидна — последующая инструментальная реализация модели может оказаться невозможной, если не следовать оригинальным правилам выделения текста в языке UML. Это правило не распространяется на отдельные примеры и фрагменты диаграмм, которые приводятся в тексте книги с чисто иллюстративными целями и лишь раскрывают особенности использования стандартных элементов языка UML.

Примечание

Приведенные дополнительные рекомендации не противоречат оригинальным правилам языка UML, а только уточняют рамки использования естественного языка при построении моделей и при описании самого языка. Поскольку описание семантики любого формального языка связано с проблемой его интерпретации, полностью обойтись без естественного языка не представляется возможным. Если вопросы использования оригинальных терминов при построении логических и физических моделей не вызывают сомнений у большинства программистов, то процесс построения концептуальных моделей сложных систем формализован в меньшей степени. Именно по этой причине исходные требования к системе формулируются на естественном для разработчиков языке (в нашем случае, на русском). В любом случае эти аспекты использования языков при построении моделей многогранны и могут служить темой отдельной работы.

В рамках языка UML все представления о модели сложной системы фиксируются в виде специальных графических конструкций, получивших название *диаграмм*. В терминах языка UML определены следующие виды диаграмм:

- диаграмма вариантов использования (use case diagram)
- диаграмма классов (class diagram)
- диаграммы поведения (behavior diagrams)
- диаграммы взаимодействия (interaction diagrams)
- диаграмма кооперации (collaboration diagram)
- диаграмма последовательности (sequence diagram)
- диаграмма состояний (statechart diagram)
- диаграмма деятельности (activity diagram)

- диаграммы реализации (implementation diagrams)
- диаграмма компонентов (component diagram)
- диаграмма развертывания (deployment diagram)

Из перечисленных диаграмм некоторые служат для обозначения двух и более других подвидов диаграмм. При этом в качестве самостоятельных представлений в языке UML используются следующие диаграммы.

1. Диаграмма вариантов использования *(глава 4)*.
2. Диаграмма классов *(глава 5)*.
3. Диаграмма кооперации *(глава 6)*.
4. Диаграмма последовательности *(глава 7)*.
5. Диаграмма состояний *(глава 8)*.
6. Диаграмма деятельности *(глава 9)*.
7. Диаграмма компонентов *(глава 10)*.
8. Диаграмма развертывания *(глава 11)*.

Перечень этих диаграмм и их названия являются каноническими в том смысле, что представляют собой неотъемлемую часть графической нотации языка UML. Более того, процесс ООАП неразрывно связан с процессом построения этих диаграмм. При этом совокупность построенных таким образом диаграмм является самодостаточной в том смысле, что в них содержится вся информация, которая необходима для реализации проекта сложной системы.

Каждая из этих диаграмм детализирует и конкретизирует различные представления о модели сложной системы в терминах языка UML. При этом диаграмма вариантов использования представляет собой наиболее общую концептуальную модель сложной системы, которая является исходной для построения всех остальных диаграмм. Диаграмма классов является, по своей сути, логической моделью, отражающей статические аспекты структурного построения сложной системы.

Диаграммы поведения также являются разновидностями логической модели, которые отражают динамические аспекты функционирования сложной системы. И, наконец, диаграммы реализации служат для представления физических компонентов сложной системы и поэтому относятся к ее физической модели. Таким образом, интегрированная модель сложной системы в нотации UML может быть представлена в виде совокупности указанных выше диаграмм (рис. 3.10).

Примечание

В ранней литературе по UML в качестве отдельной диаграммы рассматривалась еще *диаграмма объектов*. Однако в версии 1.5 она не включена в перечень канонических диаграмм, поскольку ее элементы могут присутствовать

на диаграммах других типов. Описание отдельных элементов диаграммы объектов рассматривается далее при изучении диаграммы кооперации в *главе* 6.

Рис. 3.10. Интегрированная модель сложной системы в нотации UML

3.6. Особенности изображения диаграмм языка UML

Большинство перечисленных ранее диаграмм являются в своей основе графами специального вида, состоящими из вершин в форме геометрических фигур, которые связаны между собой ребрами или дугами. Поскольку информация, которую содержит в себе граф, имеет в основном топологический характер, ни геометрические размеры, ни расположение элементов диаграмм (за некоторыми исключениями, такими как диаграмма последовательностей с метрической осью времени) не имеют принципиального значения.

Для диаграмм языка UML существуют три типа визуальных графических обозначений, которые важны с точки зрения заключенной в них информации.

- Геометрические фигуры на плоскости, играющие роль вершин графов соответствующих диаграмм. При этом сами геометрические фигуры выступают в роли *графических примитивов* языка UML, а форма этих фигур (прямоугольник, эллипс) должна строго соответствовать изображению отдельных элементов языка UML (класс, вариант использования, состояние, деятельность). Графические примитивы языка UML имеют фиксированную

семантику, переопределять которую пользователям не допускается. Графические примитивы должны иметь собственные имена, а возможно, и другой текст, который содержится внутри границ соответствующих геометрических фигур или, как исключение, вблизи этих фигур.

- Графические взаимосвязи, которые представляются различными линиями на плоскости. Взаимосвязи в языке UML обобщают понятие дуг и ребер из теории графов, но имеют менее формальный характер и более развитую семантику.
- Специальные графические символы, изображаемые вблизи от тех или иных визуальных элементов диаграмм и имеющие характер дополнительной спецификации (украшений).

Примечание

Все диаграммы в языке UML изображаются с использованием фигур на плоскости. Однако некоторые из фигур (например, кубы) могут представлять собой двумерные проекции трехмерных геометрических тел, но и в этом случае они рисуются как фигура на плоскости. Хотя в ближайшее время предполагают включить в язык UML пространственные диаграммы, в рассматриваемой версии языка такая возможность отсутствует.

Таким образом, в языке UML используется четыре основных вида конструкций.

- Графические фигуры на плоскости. Такие двумерные символы изображаются с помощью некоторых геометрических фигур и могут иметь различную высоту и ширину с целью размещения внутри этих фигур других конструкций языка UML. Наиболее часто внутри таких символов помещаются строки текста, которые уточняют семантику или фиксируют отдельные свойства соответствующих элементов языка UML. Информация, содержащаяся внутри фигур, имеет важное значение для конкретной модели проектируемой системы, поскольку регламентирует реализацию соответствующих элементов в программном коде.
- Пути, которые представляют собой последовательности из отрезков линий, соединяющих отдельные графические символы. При этом концевые точки отрезков линий должны обязательно соприкасаться с геометрическими фигурами, служащими для обозначения вершин диаграмм, как принято в теории графов *(см. главу 2)*. С концептуальной точки зрения путям в языке UML придается особое значение, поскольку они являются простыми топологическими сущностями. С другой стороны, отдельные части пути или сегменты могут не существовать сами по себе вне содержащего их пути. Пути всегда соприкасаются с другими графическими символами на обеих границах соответствующих отрезков линий. Другими словами, пути не могут обрываться на диаграмме линией, которая не соприкасается ни с одним графическим символом. Как уже отмечалось, пути могут иметь

в качестве окончания или терминатора специальную графическую фигуру — значок, который изображается на одном их концов линий.

- Значки или пиктограммы. Значок представляет собой графическую фигуру фиксированного размера и формы. Она не может увеличивать свои размеры, чтобы разместить внутри себя дополнительные символы. Значки могут размещаться как внутри других графических конструкций, так и вне их. Примерами значков могут служить окончания связей элементов диаграмм или некоторые другие дополнительные обозначения (украшения).
- Строки текста. Служат для представления различных видов информации в некоторой грамматической форме. Предполагается, что каждое использование строки текста должно соответствовать синтаксису в нотации языка UML, посредством которого может быть реализован грамматический разбор этой строки. Последний необходим для получения полной информации о модели. Например, строки текста в различных секциях обозначения класса могут соответствовать атрибутам этого класса или его операциям. На использование строк накладывается важное условие — семантика всех допустимых символов должна быть заранее определена в языке UML или служить предметом его расширения в конкретной модели.

При графическом изображении диаграмм следует придерживаться основных рекомендаций.

- Каждая диаграмма должна служить законченным представлением соответствующего фрагмента моделируемой предметной области. Речь идет о том, что в процессе разработки диаграммы необходимо учесть все сущности, важные с точки зрения контекста данной модели и диаграммы. Отсутствие тех или иных элементов на диаграмме служит признаком неполноты модели и может потребовать ее последующей доработки.
- Все сущности на диаграмме модели должны быть одного концептуального уровня. Здесь имеется в виду согласованность не только имен одинаковых элементов, но и возможность вложения отдельных диаграмм друг в друга для достижения полноты представлений. В случае достаточно сложных моделей систем желательно придерживаться стратегии последовательного уточнения или детализации отдельных диаграмм.
- Вся информация о сущностях должна быть явно представлена на диаграммах. Речь идет о том, что хотя в языке UML при отсутствии некоторых символов на диаграмме могут быть использованы их значения по умолчанию (например, в случае неявного указания видимости атрибутов и операций классов), необходимо стремиться к явному указанию свойств всех элементов диаграмм.
- Диаграммы не должны содержать противоречивой информации. Противоречивость модели может служить причиной серьезнейших проблем при ее реализации и последующем использовании на практике. Например, наличие замкнутых путей при изображении отношений агрегирования или композиции приводит к ошибкам в программном коде, который

будет реализовывать соответствующие классы. Наличие элементов с одинаковыми именами и различными атрибутами свойств, в одном пространстве имен, также приводит к неоднозначной интерпретации, и может служить источником проблем.

Примечание

Наличие в инструментальных CASE-средствах встроенной поддержки визуализации различных диаграмм языка UML позволяет в некоторой степени исключить ошибочное использование тех или иных графических символов, а также контролировать уникальность имен элементов диаграмм. Однако, поскольку вся ответственность за окончательный контроль непротиворечивости модели лежит на разработчике, необходимо помнить, что неформальный характер языка UML может служить источником потенциальных ошибок, которые в полном объеме вряд ли будут выявлены инструментальными средствами. Именно это обстоятельство требует от всех разработчиков глубокого знания нотации и семантики всех элементов языка UML.

- Диаграммы не следует перегружать текстовой информацией. Принято считать, что визуализация модели является наиболее эффективной, если она содержит минимум пояснительного текста. Как правило, наличие больших фрагментов развернутого текста служит признаком недостаточной проработанности модели или ее неоднородности, когда в рамках одной модели представляется различная по характеру информация. Поскольку общая декомпозиция модели на отдельные типы диаграмм способна удовлетворить самые детальные представления разработчиков о системе, важно уметь правильно отображать те или иные сущности и аспекты моделирования в соответствующие элементы канонических диаграмм.
- Каждая диаграмма должна быть самодостаточной для правильной интерпретации всех ее элементов и понимания семантики всех используемых графических символов. Любые пояснительные тексты, которые не являются собственными элементами диаграммы (например, комментариями), не должны приниматься во внимание разработчиками. В то же время отдельные общие фрагменты могут уточняться или детализироваться на других диаграммах этого же типа, образуя вложенные или подчиненные диаграммы. Таким образом, модель системы на языке UML представляет собой пакет иерархически вложенных диаграмм, детализация которых должна быть достаточной для последующей генерации программного кода, реализующего проект соответствующей системы.
- Количество типов диаграмм для конкретной модели приложения не является строго фиксированным. Речь идет о том, что для простых приложений нет необходимости строить все без исключения типы диаграмм. Некоторые из них могут просто отсутствовать в проекте системы, и этот факт не будет считаться ошибкой разработчика. Например, модель системы может не содержать диаграмму развертывания для приложения, выполняемого локально на компьютере пользователя. Важно понимать, что перечень диаграмм зависит от специфики конкретного проекта системы.

- Любая из моделей системы должна содержать только те элементы, которые определены в нотации языка UML. Имеется в виду требование начинать разработку проекта, используя только те конструкции, которые уже определены в метамодели UML. Как показывает практика, этих конструкций вполне достаточно для представления большинства типовых проектов программных систем. И только в случае отсутствия необходимых базовых элементов языка UML следует использовать механизмы их расширения для адекватного представления конкретной модели системы. При этом не допускается переопределение семантики тех элементов, которые отнесены к базовой нотации метамодели языка UML.

Примечание

Как не вспомнить в этой связи известный афоризм, получивший название "бритва Оккама". Суть изречения средневекового ученого-схоласта в достаточно вольном переводе сводится к следующему: "Не плоди рассуждений больше сущности". Другими словами, нужно стремиться дополнительно не усложнять и без того сложные модели систем, а по возможности упрощать их за счет унификации обозначений и семантики базовых элементов.

Процесс построения отдельных типов диаграмм имеет свои особенности, которые тесно связаны с семантикой элементов этих диаграмм. Сам процесс ООАП в контексте языка UML получил специальное название — *рациональный унифицированный процесс* (Rational Unified Process, RUP). Концепция RUP и основные его элементы разработаны А. Джекобсоном в ходе его работы над языком UML.

Примечание

При дословном переводе термина RUP теряется некоторая дополнительная семантическая окраска, связанная с двусмысленным толкованием английского Rational. Речь идет о другом варианте перевода — унифицированный процесс от фирмы Rational Software, сотрудниками которой являются с некоторых пор его разработчики, включая упомянутого выше А. Джекобсона.

Суть концепции RUP заключается в последовательной декомпозиции или разбиении процесса ООАП на отдельные этапы, на каждом из которых осуществляется разработка соответствующих типов канонических диаграмм модели системы. При этом на начальных этапах RUP строятся логические представления статической модели структуры системы, затем — логические представления модели поведения, и лишь после этого — физические представления модели системы. Как нетрудно заметить, в результате RUP должны быть построены канонические диаграммы на языке UML, при этом последовательность их разработки может соответствовать их последовательной нумерации. Таким образом, порядок изложения канонических диаграмм во второй части книги не является случайным, а определяется общими рекомендациями рационального унифицированного процесса.

Часть II

Диаграммы концептуального, логического и физического моделирования

Глава 4. Диаграмма вариантов использования (use case diagram)

Глава 5. Диаграмма классов (class diagram)

Глава 6. Диаграмма кооперации (collaboration diagram)

Глава 7. Диаграмма последовательности (sequence diagram)

Глава 8. Диаграмма состояний (statechart diagrams)

Глава 9. Диаграмма деятельности (activity diagram)

Глава 10. Диаграмма компонентов (component diagram)

Глава 11. Диаграмма развертывания (deployment diagram)

Эта часть книги посвящена систематическому изучению канонических диаграмм, которые являются основой всех модельных представлений в языке UML. Поскольку процесс построения диаграмм предполагает следование методологии рационального унифицированного процесса (RUP), последовательность изучения диаграмм не является случайной, а соответствует рекомендациям RUP. В этом случае наиболее естественно начать рассмотрение с диаграммы вариантов использования, которая служит исходным этапом разработки всех модельных представлений в нотации UML.

Глава 4

Диаграмма вариантов использования (use case diagram)

Визуальное моделирование с использованием нотации UML можно представить как некоторый процесс поуровневого спуска от наиболее общей и абстрактной концептуальной модели исходной системы к логической, а затем и к физической модели соответствующей программной системы. Для достижения этих целей вначале строится модель в форме так называемой *диаграммы вариантов использования* (use case diagram), которая описывает функциональное назначение системы или, другими словами, то, что система будет делать в процессе своего функционирования. Диаграмма вариантов использования является исходным концептуальным представлением или концептуальной моделью системы в процессе ее проектирования и разработки.

Разработка диаграммы вариантов использования преследует такие цели:

- определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы;
- сформулировать общие требования к функциональному поведению проектируемой системы;
- разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей;
- подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть данной диаграммы состоит в следующем: проектируемая система представляется в форме так называемых вариантов использования, с которыми взаимодействуют некоторые внешние сущности или актеры. При этом актером или действующим лицом называется любой объект, субъект или система, взаимодействующая с моделируемой системой извне. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь вариант использования

служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой и собственно выполнение вариантов использования.

Рассматривая диаграмму вариантов использования в качестве модели системы, можно ассоциировать ее с моделью "черного ящика" (см. рис. 1.7). Действительно, подробная детализация данной диаграммы на начальном этапе проектирования скорее имеет отрицательный характер, поскольку предопределяет способы реализации поведения системы. А согласно рекомендациям RUP, именно эти аспекты должны быть скрыты от разработчика на диаграмме вариантов использования.

В самом общем случае, диаграмма вариантов использования представляет собой граф специального вида, который является графической нотацией для представления конкретных вариантов использования, актеров и отношений между этими элементами. При этом отдельные элементы диаграммы иногда заключают в прямоугольник, который обозначает проектируемую систему в целом. Следует отметить, что отношениями данного графа могут быть только некоторые фиксированные типы взаимосвязей между актерами и вариантами использования, которые в совокупности описывают сервисы или функциональные требования к моделируемой системе.

Примечание

Как было отмечено в *главе 3*, рациональный унифицированный процесс разработки модели сложной системы представляет собой разбиение ее на составные части с минимумом взаимных связей на основе выделения пакетов. В самом языке UML пакет Варианты использования входит в состав пакета Элементы поведения. Последний специфицирует понятия, при помощи которых определяют функциональность моделируемых систем. Элементы пакета вариантов использования являются первичными по отношению к тем, с помощью которых могут быть описаны сущности, такие как системы и подсистемы. Однако внутренняя структура этих сущностей никак не описывается.

Базовые элементы диаграммы вариантов использования — собственно вариант использования и актер. С этих понятий мы и приступим к изучению диаграмм вариантов использования.

4.1. Вариант использования

Вариант использования (use case) представляет собой спецификацию общих особенностей поведения или функционирования моделируемой системы без рассмотрения внутренней структуры этой системы. Хотя каждый вариант использования определяет некоторую последовательность действий, которые

должны быть выполнены проектируемой системой при взаимодействии ее с соответствующим актером, сами эти действия не изображаются на рассматриваемой диаграмме.

Примечание

Вообще говоря, в литературе можно встретить различные переводы use case, например — прецедент, функция или просто транслитерация *юзкейс*. Увы, здесь мы сталкиваемся с парадоксальной ситуацией, когда русскоязычная терминология языка UML оказывается далеко не унифицированной. Впрочем, не углубляясь в лингвистический анализ корректности перевода отдельных терминов, автор оставляет право выбора наиболее удобного варианта перевода за читателем.

Содержание варианта использования может быть представлено в форме дополнительного пояснительного текста, который раскрывает смысл или семантику выполняемых действий при выполнении данного варианта использования. Такой пояснительный текст получил название *текста-сценария* или *сценария*. Далее в этой главе рассматривается один из шаблонов, который может быть рекомендован для написания сценариев вариантов использования (см. табл. 4.1).

Отдельный вариант использования обозначается на диаграмме эллипсом, внутри которого содержится его краткое название или имя в форме существительного (рис. 4.1, *а*) или глагола (рис. 4.1, *б*) с пояснительными словами. Сам текст должен начинаться с заглавной буквы.

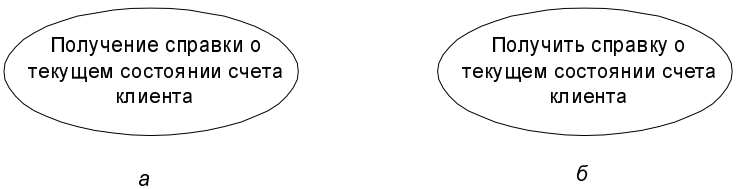


Рис. 4.1. Графическое обозначение варианта использования

Цель определения варианта использования заключается в том, чтобы зафиксировать некоторый аспект или фрагмент поведения проектируемой системы без указания особенностей реализации данной функциональности. В этом смысле каждый вариант использования соответствует отдельному сервису, который предоставляет моделируемая система по запросу актера, т. е. определяет один из способов применения системы. Сервис, который инициализируется по запросу актера, должен представлять собой законченную последовательность действий. Это означает, что после того как система закончит обработку запроса актера, она должна возвратиться в исходное состояние, в котором она готова к выполнению следующих запросов.

Диаграмма вариантов использования должна содержать конечное множество вариантов использования, которые в целом определяют все возможные стороны ожидаемого поведения системы. Для удобства множество вариантов использования может рассматриваться как отдельный пакет.

С системно-аналитической точки зрения варианты использования могут применяться как для спецификации функциональных требований к проектируемой системе, так и для документирования процесса поведения уже существующей системы. Кроме этого, варианты использования неявно специфицируют требования, определяющие особенности взаимодействия пользователей с системой, чтобы иметь возможность корректно работать с предоставляемыми данной системой сервисами.

Применение вариантов использования на всех этапах работы над проектом позволяет не только достичь требуемого уровня унификации обозначений для представления функциональности подсистем и системы в целом, но и является мощным средством последовательного уточнения требований к проектируемой системе на основе их итеративного обсуждения со всеми заинтересованными специалистами.

В метамодели UML вариант использования является подклассом классификатора, который описывает последовательности действий, выполняемых отдельным экземпляром варианта использования. Эти действия включают изменения состояния и взаимодействия со средой варианта использования. Сами последовательности выполняемых действий могут описываться различными способами, включая рассматриваемые далее в книге графы деятельности и конечные автоматы.

Примерами вариантов использования могут являться следующие действия: проверка состояния текущего счета клиента, оформление заказа на покупку товара, получение дополнительной информации о кредитоспособности клиента, отображение графической формы на экране монитора и другие действия.

4.2. Актеры

Актер (actor) представляет собой любую внешнюю по отношению к моделируемой системе сущность, которая взаимодействует с системой и использует ее функциональные возможности для достижения определенных целей или решения частных задач. При этом актеры служат для обозначения согласованного множества ролей, которые могут играть пользователи в процессе взаимодействия с проектируемой системой. Каждый актер может рассматриваться как некая отдельная роль относительно конкретного варианта использования. Стандартным графическим обозначением актера на диаграммах является фигурка "человечка", под которой записывается конкретное имя актера (рис. 4.2).

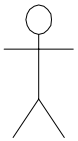


Рис. 4.2. Графическое обозначение актера

В некоторых случаях актер может обозначаться в виде прямоугольника класса с ключевым словом (стереотипом) <<актер>> и обычными составляющими элементами класса. Имена актеров должны начинаться с заглавной буквы и следовать рекомендациям использования имен для типов и классов модели. При этом символ отдельного актера связывает соответствующее описание актера с конкретным именем.

Примечание

Имя актера должно быть достаточно информативным с точки зрения семантики. Вполне подходят для этой цели наименования должностей в компании (например, продавец, кассир, менеджер, президент). Не рекомендуется давать актерам имена собственные (например, "О. Бендер") или моделей конкретных устройств (например, "маршрутизатор Cisco 3640"), даже если это с очевидностью следует из контекста проекта. Дело в том, что одно и то же лицо может выступать в нескольких ролях и, соответственно, обращаться к различным сервисам системы. Например, посетитель банка может являться потенциальным клиентом, и тогда он востребует один из его сервисов, а может быть налоговым инспектором или следователем прокуратуры. Сервис для последнего может быть совершенно исключительным по своему характеру.

Примерами актеров могут быть: кассир, клиент банка, банковский служащий, президент, продавец магазина, менеджер отдела продаж, пассажир авиарейса, водитель автомобиля, администратор гостиницы, сотовый телефон и другие сущности, имеющие отношение к концептуальной модели соответствующей предметной области.

Примечание

В метамодели актер является подклассом классификатора. Актеры могут взаимодействовать с множеством вариантов использования и иметь множество интерфейсов, каждый из которых может представлять особенности взаимодействия других элементов с отдельными актерами.

Актеры используются для моделирования внешних по отношению к проектируемой системе сущностей, которые взаимодействуют с системой и используют ее в качестве отдельных пользователей. В качестве актеров могут выступать другие системы, в том числе подсистемы проектируемой системы или ее отдельные классы. Важно понимать, что каждый актер определяет

некоторое согласованное множество ролей, в которых могут выступать пользователи данной системы в процессе взаимодействия с ней. В каждый момент с системой взаимодействует вполне определенный пользователь, при этом он играет или выступает в одной из таких ролей. Наиболее наглядный пример актера — конкретный посетитель Web-сайта в Интернете, со своими собственными параметрами аутентификации.

Любой объект или субъект, который согласуется с подобным неформальным определением актера, представляет собой экземпляр или пример актера. Для моделируемой системы актерами могут быть как субъекты-пользователи, так и другие системы. Поскольку пользователи моделируемой системы всегда являются внешними по отношению к ней, то они всегда представляются в виде актеров.

Поскольку, в общем случае, актер всегда находится вне системы, его внутренняя структура никак не определяется. Для актера имеет значение только его внешнее представление, т. е. то, как он воспринимается со стороны системы. Актеры взаимодействуют с системой посредством передачи и приема сообщений от вариантов использования. Сообщение представляет собой запрос актером сервиса от системы и получение этого сервиса. Это взаимодействие может быть выражено посредством ассоциаций между отдельными актерами и вариантами использования. Кроме этого, с актерами могут быть связаны интерфейсы, которые определяют, каким образом другие элементы модели взаимодействуют с этими актерами.

Два и более актера могут иметь общие свойства, т. е. взаимодействовать с одним и тем же множеством вариантов использования одинаковым образом. Такая общность свойств и поведения представляется в виде рассматриваемого ниже отношения обобщения с другим, возможно, абстрактным актером, который моделирует соответствующую общность ролей. Совокупность отношений, которые могут присутствовать на диаграмме вариантов использования, будет рассмотрена ниже в данной главе.

Для идентификации актеров в процессе проектирования системы могут быть рекомендованы вопросы, ответы на которые должны помочь разработчикам на начальных этапах выполнения проекта.

1. Какие организации или лица будут использовать проектируемую систему?
2. Кто будет получать пользу от использования системы?
3. Кто будет использовать информацию от системы?
4. Будет ли использовать система внешние ресурсы?
5. Может ли один пользователь играть несколько ролей при взаимодействии с системой?
6. Могут ли различные пользователи играть одну роль при взаимодействии с системой?
7. Будет ли система взаимодействовать с законодательными, исполнительными, налоговыми или другими органами?

Прежде чем перейти к рассмотрению отношений, изображаемых на диаграммах вариантов использования, следует остановиться на общих для всех конструкциях, которыми служат примечания.

4.3. Примечания

Примечание (note) в языке UML предназначено для включения в модель произвольной текстовой информации, имеющей непосредственное отношение к контексту разрабатываемого проекта. В качестве такой информации могут быть комментарии разработчика (например, дата и версия разработки диаграммы или ее отдельных компонентов), ограничения (например, на значения отдельных связей или экземпляры сущностей) и помеченные значения. Применительно к диаграммам вариантов использования примечание может иметь уточняющую информацию, относящуюся к контексту тех или иных вариантов использования.

Графически примечания на всех типах диаграмм обозначаются прямоугольником с "загнутым" верхним правым уголком (рис. 4.3). Собственно, текст примечания размещается внутри этого прямоугольника. Примечание может относиться к любому элементу диаграммы, в этом случае их соединяет пунктирная линия. Если примечание относится к нескольким элементам, то от него проводятся, соответственно, несколько линий. Как уже отмечалось, примечания могут присутствовать не только на диаграмме вариантов использования, но и на других канонических диаграммах.

Рис. 4.3. Примеры примечаний в языке UML

Если в примечании указывается ключевое слово <<constraint>>, то данное примечание является ограничением, налагаемым на соответствующий элемент модели, но не на саму диаграмму. При этом запись ограничения заключается в фигурные скобки и должна соответствовать правилам правильного

построения выражений языка OCL. Более подробно язык объектных ограничений и примеры его использования будут рассмотрены в *Приложении*. Однако, для диаграмм вариантов использования, включать ограничения в модели не рекомендуется, поскольку они достаточно жестко регламентируют физические аспекты реализации системы. Подобная регламентация противоречит концептуальному характеру модели вариантов использования.

4.4. Отношения на диаграмме вариантов использования

Между элементами диаграммы вариантов использования могут существовать различные отношения, которые описывают взаимодействие экземпляров одних актеров и вариантов использования с экземплярами других. Один актер может взаимодействовать с несколькими вариантами использования. В этом случае этот актер обращается к нескольким сервисам данной системы. В свою очередь один вариант использования может взаимодействовать с несколькими актерами, предоставляя для всех них свой сервис.

В то же время два варианта использования, определенные в рамках одной моделируемой системы, также могут взаимодействовать друг с другом, однако характер этого взаимодействия будет отличаться от взаимодействия с актерами. Однако в обоих случаях способы взаимодействия элементов модели предполагают некоторый обмен сигналами или сообщениями, которые инициируют реализацию функционального поведения моделируемой системы.

В языке UML имеется несколько стандартных видов отношений между актерами и вариантами использования:

- отношение ассоциации (association relationship);
- отношение включения (include relationship);
- отношение расширения (extend relationship);
- отношение обобщения (generalization relationship).

При этом общие свойства вариантов использования могут быть представлены тремя различными способами, а именно с помощью отношений включения, расширения и обобщения.

4.4.1. Отношение ассоциации

Отношение ассоциации является одним из фундаментальных понятий в языке UML и в той или иной степени используется при построении всех графических моделей систем в форме канонических диаграмм.

Применительно к диаграммам вариантов использования ассоциация служит для обозначения специфической роли актера при взаимодействии с отдельным

вариантом использования. Другими словами, ассоциация специфицирует семантические особенности взаимодействия актеров и вариантов использования в графической модели системы. На диаграмме вариантов использования, так же как и на других диаграммах, отношение ассоциации обозначается сплошной линией между актером и вариантом использования. Отношение ассоциации может иметь собственное имя, а концевые точки ассоциации — кратность (рис. 4.4).

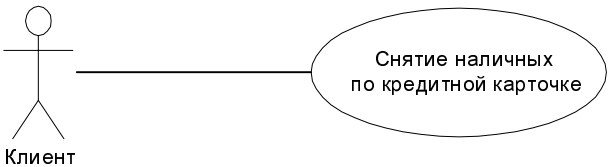


Рис. 4.4. Пример графического представления отношения ассоциации между актером и вариантом использования

В контексте диаграммы вариантов использования отношение ассоциации между актером и вариантом использования может указывать на тот факт, что актер является инициатором соответствующего варианта использования. В этом случае такого актера часто называют *главным актером*. В других случаях подобная ассоциация может указывать на актера, которому предоставляется справочная информация о результатах функционирования моделируемой системы. Таких актеров часто называют *второстепенными*. Более детальное описание семантических особенностей отношения ассоциации будет дано при рассмотрении других диаграмм в последующих главах книги.

4.4.2. Отношение включения

Отношение включения устанавливается только между двумя вариантами использования и указывает на тот факт, что некоторое заданное поведение для одного варианта использования включается в качестве составного фрагмента в последовательность поведения другого. Данное отношение является направленным бинарным отношением в том смысле, что пара экземпляров вариантов использования всегда упорядочена в отношении включения.

Отношение включения, направленное от варианта использования *А* к варианту использования *Б*, указывает, что каждый экземпляр *А* включает в себя функциональное поведение, заданное для *Б*. Эти свойства дополняют поведение соответствующего варианта *А* на данной диаграмме. Графически данное отношение обозначается пунктирной линией со стрелкой (отношение зависимости), направленной от базового варианта использования к включаемому. При этом данная линия со стрелкой помечается стереотипом <<include>> (включает), как показано на рис. 4.5.

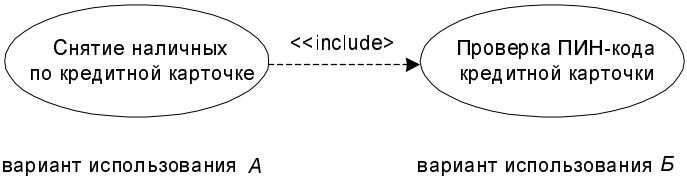


Рис. 4.5. Пример графического изображения отношения включения между вариантами использования

Семантика этого отношения определяется следующим образом. Процесс выполнения базового варианта использования (*А*) включает в себя, как собственное подмножество, последовательность действий, которая определена для включаемого варианта использования (*Б*). При этом выполнение включаемой последовательности действий происходит всегда при инициировании базового варианта использования.

Один вариант использования может быть включен в несколько других вариантов, а также включать в себя другие варианты. Включаемый вариант использования может быть независимым от базового варианта в том смысле, что он предоставляет последнему некоторое инкапсулированное поведение, детали реализации которого скрыты от последнего и могут быть легко перераспределены между несколькими включаемыми вариантами использования. Более того, базовый вариант зависит только от результатов выполнения включаемого в него варианта использования, но не от структуры включаемых в него вариантов.

4.4.3. Отношение расширения

Отношение расширения определяет взаимосвязь базового варианта использования с некоторым другим вариантом использования, функциональное поведение которого задействуется базовым не всегда, а только при выполнении некоторых дополнительных условий. В метамодели языка UML отношение расширения является зависимостью, направленной к базовому варианту использования и соединенной с ним в так называемой точке расширения.

Отношение расширения между вариантами использования обозначается пунктирной линией со стрелкой (отношение зависимости), направленной от того варианта использования, который является расширением для базового варианта использования. Данная линия со стрелкой должна быть помечена стереотипом <<extend>> (расширяет), как показано на рис. 4.6.

Таким образом, если имеет место отношение расширения между базовым вариантом использования *А* и вариантом использования *Б*, то это означает, что свойства поведения варианта использования *А* в некоторых случаях могут быть дополнены за счет функциональности *Б*. Чтобы это расширение

имело место, должно быть выполнено определенное логическое условие данного отношения.

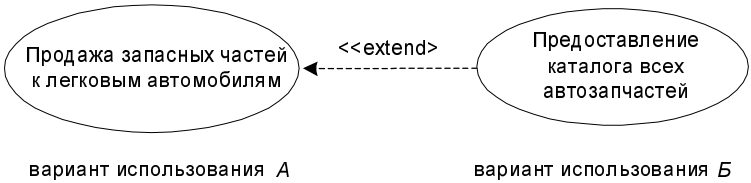


Рис. 4.6. Пример графического изображения отношения расширения между вариантами использования

Отношение расширения позволяет моделировать тот факт, что один из вариантов использования должен присоединять к своему поведению некоторое дополнительное поведение, определенное для другого варианта использования. В то же время, данное отношение всегда предполагает проверку некоторого условия и ссылку на точку расширения в базовом варианте использования. Точка расширения определяет место, в которое должно быть помещено расширение при выполнении соответствующего логического условия.

Один из вариантов использования может быть расширением для нескольких базовых вариантов, а также иметь в качестве собственных расширений несколько других вариантов. Базовый вариант использования не зависит от своих расширений.

Семантика отношения расширения определяется следующим образом. Если базовый вариант использования выполняет некоторую последовательность действий, которая определяет его поведение, и при этом имеется точка расширения на экземпляр другого варианта использования, которая является первой из всех точек расширения у базового варианта, то проверяется логического условие данного отношения. Если это условие выполняется, исходная последовательность действий расширяется посредством включения действий другого варианта использования. Следует заметить, что условие отношения расширения проверяется лишь один раз — при первой ссылке на точку расширения, и если оно выполняется, то все расширяющие варианты использования вставляются в базовый вариант.

В представленном выше примере (см. рис. 4.6) фрагмента диаграммы вариантов использования системы продажи запасных частей к легковым автомобилям только в некоторых случаях может потребоваться предоставление клиенту каталога всех запасных частей. При этом условием расширения является запрос от клиента на получение каталога автозапчастей. Очевидно, что после получения каталога клиенту может потребоваться некоторое время на его изучение, в течение которого процесс продажи приостанавливается. После ознакомления с каталогом клиент решает либо в пользу приобретения некоторых комплектующих, либо отказывается от покупки вообще.

Базовый вариант использования может иметь как единственную точку расширения, так и множество отдельных точек. Такие точки расширения могут быть специфицированы различными способами, например, с помощью текста примечания на естественном языке (рис. 4.7), пред- и постусловий, а также с использованием имен состояний в автомате.

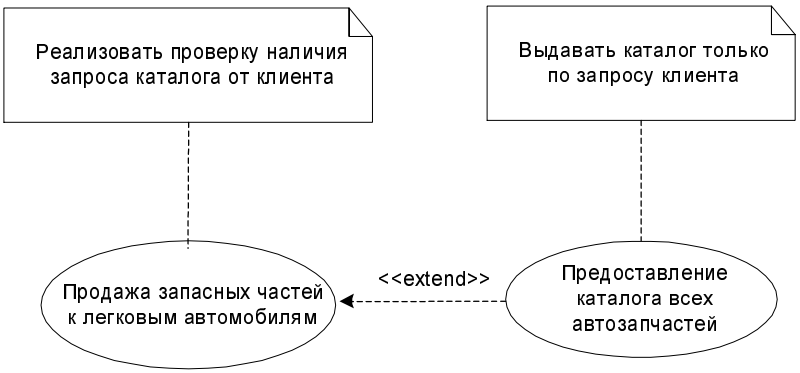


Рис. 4.7. Графическое изображение отношения расширения с примечаниями условий выполнения вариантов использования

Примечание

Следует отметить, что хотя в языке UML имеется механизм точек расширения вариантов использования, на практике он не нашел широкого применения. Причиной является нежелание многих разработчиков излишне усложнять диаграммы вариантов использования логическими условиями. Это вполне оправдано, поскольку для представления сложной логики поведения моделируемых систем больше подходят другие типы канонических диаграмм, которые будут рассматриваться далее.

4.4.4. Отношение обобщения

Отношение обобщения служит для указания того факта, что некоторый вариант использования *А* является специальным случаем варианта использования *Б*. В этом случае вариант *А* будет являться специализацией варианта *Б*. При этом *Б* называется *предком* или родителем по отношению *А*, а вариант *А* называется *потомком* по отношению к варианту использования *В*. Следует подчеркнуть, что потомок наследует все свойства поведения своего родителя, а также может обладать дополнительными особенностями поведения.

Графически отношение обобщения обозначается сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на родительский вариант использования *Б* (рис. 4.8). Эта линия со стрелкой имеет специальное название — *стрелка-обобщение*.

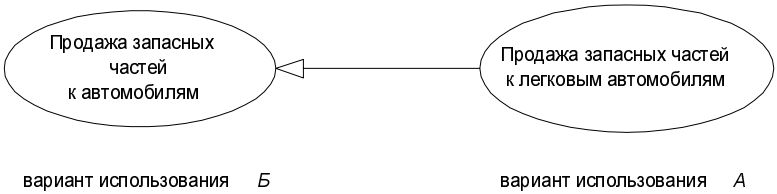


Рис. 4.8. Пример графического изображения отношения обобщения между вариантами использования

Отношение обобщения между вариантами использования применяется в том случае, когда необходимо отметить, что дочерние варианты использования обладают всеми особенностями поведения родительских вариантов. При этом дочерние варианты использования участвуют во всех отношениях родительских вариантов. В свою очередь, дочерние варианты могут наделяться новыми свойствами поведения, которые отсутствуют у родительских вариантов использования, а также уточнять или модифицировать наследуемые от них свойства поведения.

Применительно к данному отношению, один вариант использования может иметь несколько родительских вариантов. В этом случае реализуется множественное наследование свойств и поведения отношения предков. С другой стороны, один вариант использования может быть предком для нескольких дочерних вариантов, что соответствует таксономическому характеру отношения обобщения.

Между отдельными актерами также может существовать отношение обобщения. Данное отношение является направленным и указывает на факт специализации одних актеров относительно других. Например, отношение обобщения от актера *А* к актеру *Б* отмечает тот факт, что каждый экземпляр актера *А* является одновременно экземпляром актера *Б* и обладает всеми его свойствами. В этом случае актер *Б* является родителем по отношению к актеру *А,* а актер *А*, соответственно, потомком актера *В*. При этом актер *А* обладает способностью играть такое же множество ролей, что и актер *В*. Графически данное отношение также обозначается стрелкой обобщения, т. е. сплошной линией со стрелкой в форме не закрашенного треугольника, которая указывает на родительского актера (рис. 4.9).

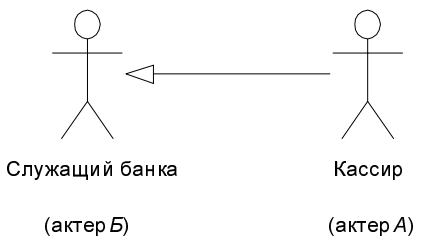


Рис. 4.9. Пример графического изображения отношения обобщения между актерами

В завершение рассмотрения отношений на диаграммах вариантов использования следует заметить, что рассмотренные три последних отношения могут существовать только между вариантами использования, которые определены для одной и той же моделируемой системы. Причина этого заключается в том, что поведение рассматриваемой моделируемой системы обусловлено вариантами использования только этой системы. Другими словами, все варианты использования выполняются только внутри данной системы. Если некоторый вариант использования должен иметь отношение обобщения, включения или расширения с вариантом использования другой моделируемой системы, получаемые в результате вариантов должны быть включены в обе системы, что противоречит семантическим правилам представления элементов языка UML. Однако эти отношения, определенные в пределах одной системы, могут быть использованы в пределах другой системы, если обе системы являются подсистемами некоторой более общей системы или связаны между собой отношением обобщения.

4.5. Расширение языка UML для бизнес-моделирования

Как было отмечено ранее в *главе 3*, спецификация языка UML включает в себя специальные механизмы расширения, которые позволяют ввести в рассмотрение дополнительные графические обозначения, ориентированные на решение задач из определенной предметной области. Примеры подобных обозначений, которые отсутствуют в исходной документации на язык UML, содержатся в известном программном инструментарии IBM Rational Rose 2002. Хотя детально особенности этого CASE-средства рассматриваются в *части III*, в данный момент имеет смысл остановиться на тех графических примитивах, которые могут быть изображены на диаграммах вариантов использования.

В рамках IBM Rational Rose 2002 предложено несколько десятков специальных графических примитивов, которые могут быть использованы при построении различных диаграмм. Из них при построении диаграмм вариантов использования наибольшее применение нашли следующие графические элементы.

- *Бизнес-актер* (business actor) — индивидуум, группа, организация, компания или система, которые взаимодействуют с моделируемой бизнес-системой, но не входят в нее, т. е. не являются частью моделируемой системы. Графическое изображение бизнес-актера приводится на рис. 4.10, *а*. Примерами бизнес-актеров являются клиенты, покупатели, поставщики, партнеры.
- *Сотрудник* (business worker) — индивидуум, который действует внутри моделируемой бизнес-системы, взаимодействует с другими сотрудниками и является участником бизнес-процесса. Графическое изображение сотрудника приводится на рис. 4.10, *б*.

- *Бизнес-вариант использования* (business use case) — определяющий функциональность моделируемой системы, которая ориентирована на выполнение отдельного бизнес-процесса. Графическое изображение бизнес-варианта использования приводится на рис. 4.10, *в*.

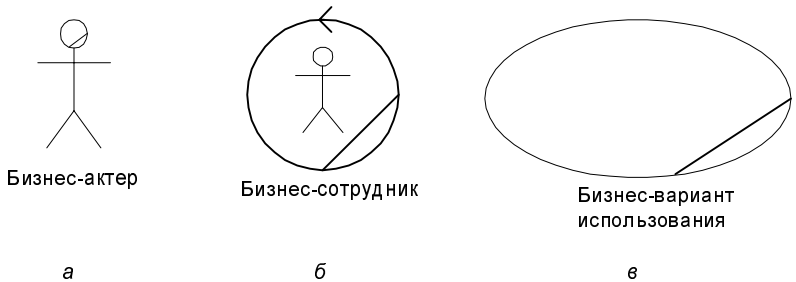


Рис. 4.10. Графические изображения бизнес-актера (*а*), сотрудника (*б*) и бизнес-варианта использования (*в*)

Для иллюстрации применения этих элементов графической нотации рассмотрим пример представления диаграмм вариантов использования для системы продажи товаров по каталогу двумя способами. Эта модель может быть использована при создании и автоматизации соответствующих информационных систем.

В качестве актеров рассматриваемой системы могут выступать два субъекта, один из которых является продавцом, а другой — покупателем. Каждый из этих актеров взаимодействует с рассматриваемой системой продажи товаров по каталогу и является ее пользователем, т. е. они оба обращаются к соответствующему сервису "Оформление заказа на покупку товара". Как следует из существа выдвигаемых к системе требований, этот сервис выступает в качестве базового варианта использования рассматриваемой системы.

С одной стороны, продажа товаров предполагает согласование условий оплаты с покупателем и заказ товара со склада. Поскольку эта функциональность выполняется всегда, она может быть выделена в отдельные варианты использования, которые будут связаны с базовым отношением включения.

С другой стороны, продажа товаров может предполагать наличие самостоятельного информационного объекта — каталога товаров, который в некотором смысле не зависит от реализации сервиса по обслуживанию покупателей. В нашем случае, каталог товаров может запрашиваться покупателем или продавцом при необходимости выбора товара и уточнения его свойств. Вполне резонно представить сервис "Предоставление каталога товаров" в качестве самостоятельного варианта использования.

Дальнейшая детализация модели может быть выполнена на основе установления дополнительных отношений типа отношения "обобщение — специализация"

для уже имеющихся компонентов диаграммы вариантов использования. Так, в рамках рассматриваемой системы продажи товаров может иметь самостоятельное значение и специфические особенности отдельная категория товаров — компьютеры. В этом случае диаграмма может быть дополнена вариантом использования "Оформление заказа на покупку компьютера", который связан с сервисом "Оформление заказа на покупку товара" отношением обобщения.

Полученная в результате диаграмма вариантов использования будет содержать 5 вариантов использования и 2-х актеров, между которыми установлены соответствующие отношения включения, расширения и обобщения. Эта диаграмма, изображенная в общих обозначениях нотации языка UML, представлена на рис. 4.11.

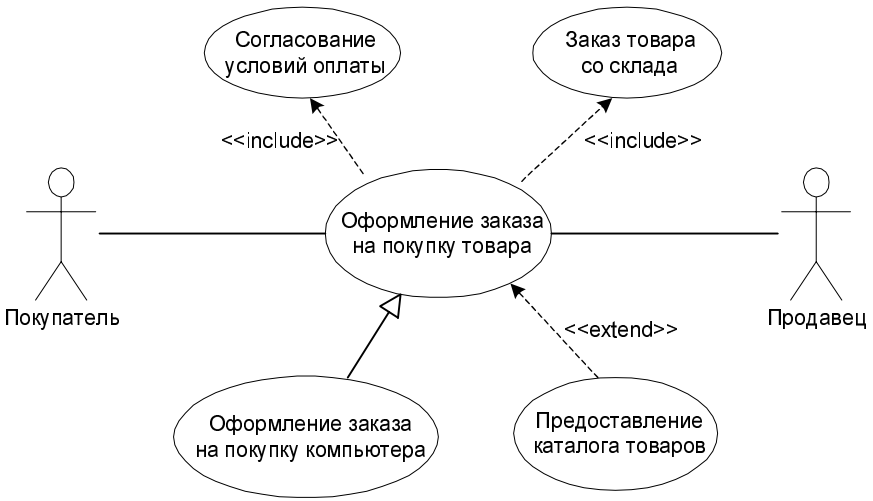


Рис. 4.11. Диаграмма вариантов использования для системы продажи товаров по каталогу в общих обозначениях языка UML

Анализируя рассматриваемую систему продажи товаров по каталогу, можно заметить, что она представляет собой типичную бизнес-систему, особенности которой связаны с получением определенной прибыли от реализации соответствующих бизнес-процессов. При этом роли покупателя и продавца в рассматриваемой системе существенно отличаются. Действительно, покупатель является внешним по отношению к системе субъектом, в то время как продавец является частью бизнес-системы.

Для более точной передачи этих особенностей покупателя можно изобразить как бизнес-актера, а продавца — как сотрудника. В свою очередь все определенные ранее варианты использования могут быть представлены

в форме бизнес-вариантов использования. Вариант этой же диаграммы, изображенной в специальных дополнительных обозначениях IBM Rational Rose 2002, представлен на рис. 4.12.

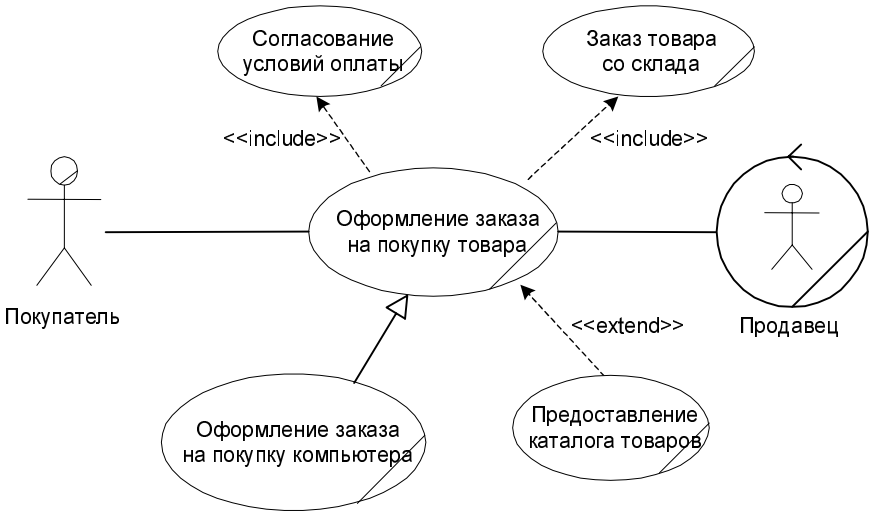


Рис. 4.12. Диаграмма вариантов использования для системы продажи товаров по каталогу в специальных дополнительных обозначениях IBM Rational Rose 2002

Хотя изображенные диаграммы вариантов использования (рис. 4.11, 4.12) тождественны по своей структуре, вторая из них более точно отражает семантику предметной области. Именно поэтому наиболее часто вторая диаграмма (рис. 4.12) изображается при выполнении рабочего процесса проектирования, называемого "Бизнес-моделированием". Этот рабочий процесс (или *дисциплина* по терминологии RUP) направлен на анализ и построение исходной концептуальной модели проектируемой бизнес-системы.

4.6. Текстовые сценарии вариантов использования

Хотя одним из требований языка UML является самодостаточность диаграмм для представления информации о моделях проектируемых систем, многие разработчики согласны с тем, что изобразительных средств языка UML явно недостаточно, чтобы учесть на диаграммах вариантов использования особенности функционального поведения достаточно сложной системы. С этой целью настоятельно рекомендуется дополнять этот тип диаграмм текстовыми сценариями, которые уточняют или детализируют последовательность действий, совершаемых системой при выполнении ее вариантов использования.

В литературе, включая базу шаблонов RUP, предложены для этой цели различные способы представления или написания подобных сценариев. Один из таких шаблонов рассматривается ниже и может быть рекомендован читателям для применения на начальных этапах концептуального моделирования (табл. 4.1).

Таблица 4.1. *Шаблон для написания сценария отдельного варианта использования*

Главный раздел	**Раздел "Типичный ход событий"**	**Раздел "Исключения"**	**Раздел "Примечания"**
Имя варианта использования Актеры Цель Краткое описание Тип Ссылки на другие варианты использования	Типичный ход событий, приводящий к успешному выполнению варианта использования	Исключение № 1 Исключение № 2 Исключение № 3	Примечания

При написании сценариев вариантов использования важно понимать, что текст сценария должен дополнять или уточнять диаграмму вариантов использования, но не заменять ее полностью. В противном случае будут потеряны достоинства визуального представления моделей.

Примечание

Увы, это замечание отнюдь не риторическое. Некоторые читатели, с оптимизмом воспринявшие работу А. Кокберна по написанию эффективных вариантов использования, склонны вообще отказаться от графического изображения последних. Судя по всему, адепты текстовых сценариев забывают о существовании в языке UML других типов диаграмм, поскольку их основным аргументом против изображения диаграмм вариантов использования является их перегруженность или переизбыток визуальных элементов. Действительно, при разработке диаграмм вариантов использования следует вовремя остановиться и не пытаться на этом типе диаграмм изображать логику или последовательность выполнения отдельных действий. Ведь для последней цели специально предназначены диаграммы деятельности, которые рассматриваются в *главе 9*. Впрочем, при работе с диаграммами вариантов использования допускаются и другие ошибки, часть из которых рассматривается в завершение этой главы.

Построение диаграммы вариантов использования является первым этапом процесса объектно-ориентированного анализа и проектирования, цель которого — представить совокупность функциональных требований к поведению

проектируемой системы. Спецификация требований к проектируемой системе в форме диаграммы вариантов использования и дополнительных сценариев может представлять собой самостоятельную модель, которая в языке UML получила название *модели вариантов использования* и имеет свое специальное стандартное имя или стереотип — <<useCaseModel>>.

В последующем все заданные в этой модели требования могут быть представлены в виде общей модели системы, которая может быть оформлена в виде отдельного пакета Система. Этот пакет в свою очередь может представлять собой иерархию пакетов, на самом верхнем уровне которых содержится множество классов, реализующих базовые варианты использования проектируемой системы. При этом пакет системы самого верхнего уровня может быть дополнительно помечен стереотипом <<topLevelPackage>>.

4.7. Пример построения диаграммы вариантов использования для системы управления банкоматом

Для иллюстрации особенностей построения конкретной диаграммы вариантов использования рассмотрим модель банкомата, например, Сбербанка РФ. Процесс функционирования этой системы хорошо знаком всем владельцам кредитных карточек, поэтому не требует пространного описания. Особенностью отечественного банкомата является отсутствие возможности перевода средств с одного счета на другой. Эта система будет использоваться и в дальнейшем, в качестве сквозного примера разработки интегрированной модели в нотации языка UML, при этом исходная модель будет дополняться новыми типами канонических диаграмм по мере их изучения в последующих главах.

Рассматриваемая система имеет двух актеров, один из которых является клиентом банкомата, а другой — Банком, который осуществляет выполнение соответствующих транзакций. Каждый из этих актеров взаимодействует с банкоматом, хотя главным актером является Клиент, поскольку именно он инициирует функциональность банкомата.

Базовыми вариантами использования является "Снятие наличных по кредитной карточке" и "Получение справки о состоянии счета". Поскольку для выполнения этих вариантов использования необходимо аутентифицировать кредитную карточку, то они обращаются к дополнительному сервису "Проверка ПИН-кода кредитной карточки". Как следует из существа выдвигаемых к банкомату требований, этот сервис может выступать в качестве отдельного варианта использования разрабатываемой диаграммы и соединяться с первыми двумя вариантами отношением включения. Первоначальная диаграмма вариантов использования может включать в себя только указанных двух актеров и три варианта использования (рис. 4.13).

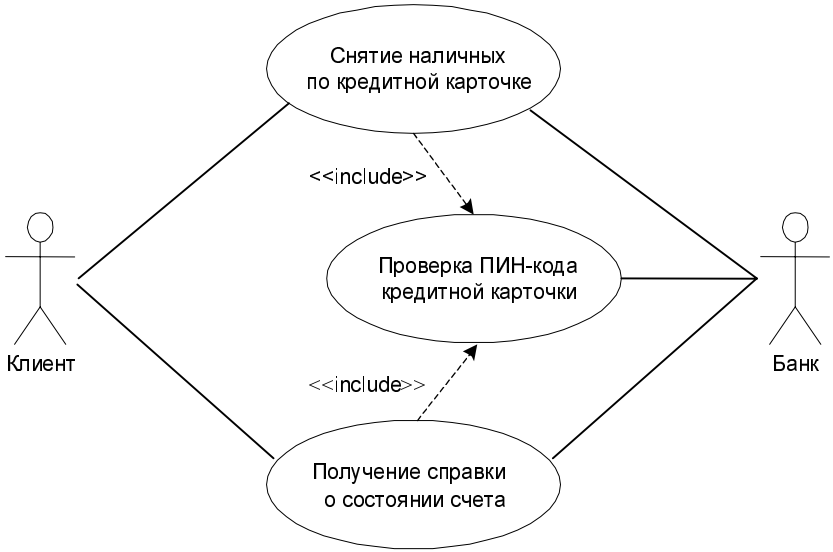


Рис. 4.13. Исходная диаграмма вариантов использования для примера разработки системы продажи товаров по каталогу

Примечание

Заинтересованные читатели могут выполнить дальнейшую детализацию построенной диаграммы, например, введя в рассмотрение еще одного актера – Оператора, осуществляющего техническое обслуживание банкомата, с дополнительным вариантом использования "Техническое обслуживание банкомата". Это вполне может служить в качестве упражнения для самостоятельной работы над материалом.

На следующем этапе разработки модели вариантов использования рассматриваемой системы напишем текстовый сценарий на основе предложенного ранее шаблона (табл. 4.1). Этот сценарий будет дополнять диаграмму, раскрывая содержание отдельных действий, выполняемых системой и актерами в процессе снятия наличных по кредитной карточке. В этом случае сценарий удобно представить в виде трех отдельных таблиц, каждая из которых описывает отдельный раздел шаблона, при этом последний раздел "Примечания" отсутствует.

В главном разделе сценария (табл. 4.2) указывается имя рассматриваемого варианта использования, имена взаимосвязанных с ним актеров, цель выполнения варианта, условный тип и ссылки на другие варианты использования.

Таблица 4.2. *Главный раздел*

Вариант использования	Снятие наличных по кредитной карточке
Актеры	Клиент, Банк
Цель	Получение требуемой суммы наличными
Краткое описание	Клиент запрашивает требуемую сумму. Банкомат обеспечивает доступ к счету клиента. Банкомат выдает клиенту наличные
Тип	Базовый
Ссылки на другие варианты использования	Включает в себя варианты использования : • проверка ПИН-кода кредитной карточки • идентифицировать кредитную карточку

В следующем разделе сценария (табл. 4.3) описывается последовательность действий, приводящая к успешному выполнению рассматриваемого варианта использования. При этом инициатором действий должен выступать один из актеров (в нашем случае — Клиент). Для удобства последующих ссылок каждое действие помечается порядковым номером в последовательности.

Таблица 4.3. *Раздел "Типичный ход событий"*

Типичный ход событий	
Действия актеров	**Отклик системы**
1. Клиент вставляет кредитную карточку в устройство чтения банкомата Исключение №1: Кредитная карточка недействительна	2. Банкомат проверяет кредитную карточку 3. Банкомат предлагает ввести ПИН-код
4. Клиент вводит персональный PIN-код Исключение №2: Клиент вводит неверный ПИН-код	5. Банкомат проверяет ПИН-код 6. Банкомат отображает опции меню
7. Клиент выбирает снятие наличных со своего счета	8. Система делает запрос в Банк и выясняет текущее состояние счета клиента 9. Банкомат предлагает ввести требуемую сумму
10. Клиент вводит требуемую сумму 11. Банк проверяет введенную сумму Исключение №3. Требуемая сумма превышает сумму на счете клиента	12. Банкомат изменяет состояние счета клиента, выдает наличные и чек

Таблица 4.3 *(окончание)*

Типичный ход событий	
Действия актеров	**Отклик системы**
13. Клиент получает наличные и чек	14. Банкомат предлагает клиенту забрать его кредитную карточку
15. Клиент получает свою кредитную карточку	16. Банкомат отображает сообщение о своей готовности к работе

Наконец, в третьем разделе сценария (табл. 4.4) описываются последовательности действий, выполняемых при возникновении исключительных ситуаций (исключений).

Таблица 4.4. *Раздел "Исключения"*

Действия актера	**Отклик системы**
Исключение №1. Кредитная карточка недействительна или неверно вставлена	
	3. Банкомат отображает информацию о неверно вставленной кредитной карточке
	14. Банкомат возвращает клиенту его кредитную карточку
15. Клиент получает свою кредитную карточку	
Исключение №2. Клиент вводит неверный ПИН-код	
	6. Банкомат отображает информацию о неверном ПИН-коде
4. Клиент вводит новый ПИН-код	
Исключение №3. Требуемая сумма превышает сумму на счете клиента	
	12. Банкомат отображает информацию о превышении кредита
10. Клиент вводит новую требуемую сумму	

Поскольку два других варианта использования рассматриваемой модели вполне тривиальны в логическом аспекте, но в техническом аспекте их реализация и может представлять собой серьезную проблему, их описание может отсутствовать в данном сценарии модели вариантов использования.

Впрочем, заинтересованные читатели могут самостоятельно дополнить написанный сценарий, описав не только варианты использования "Получение справки о состоянии счета" и "Проверка ПИН-кода кредитной карточки", но и рассмотрев дополнительные исключения, такие, как отказ клиента от получения наличных после проверки ПИН-кода и пр. Все это вполне может служить в качестве упражнения.

4.8. Рекомендации по разработке диаграмм вариантов использования

Одно из главных назначений диаграммы вариантов использования заключается в формализации функциональных требований к системе и возможности согласования полученной модели с заказчиком на ранней стадии проектирования. Любой из вариантов использования потенциально может быть подвергнут дальнейшей декомпозиции на множество подвариантов использования отдельных элементов, которые образуют исходную сущность. Однако рекомендуемое общее количество актеров в модели — не более 20, а вариантов использования — не более 50. В противном случае модель теряет свою наглядность и, возможно, заменяет собой одну из некоторых других диаграмм.

Для разработки диаграммы вариантов использования можно рекомендовать такую последовательность действий.

1. Определить главных (первичных) и второстепенных актеров.
2. Определить цели главных актеров по отношению к системе.
3. Сформулировать базовые (стратегические) варианты использования.
4. Упорядочить варианты использования по степени убывания риска их реализации.
5. Последовательно рассмотреть все базовые варианты использования в порядке убывания их степени риска.
6. Выделить участников, интересы, предусловия и постусловия выбранного варианта использования.
7. Написать успешный сценарий выполнения выбранного варианта использования.
8. Определить исключения (неуспех) в сценарии варианта использования.
9. Написать сценарии для всех исключений.
10. Выделить общие варианты использования и изобразить их взаимосвязи с базовыми со стереотипом <<include>>.
11. Выделить варианты использования для исключений и изобразить их взаимосвязи с базовыми со стереотипом <<extend>>.
12. Проверить диаграмму на отсутствие дублирования вариантов использования и актеров.

Семантика построения диаграммы вариантов использования должна определяться следующими особенностями рассмотренных ранее элементов модели. Отдельный экземпляр варианта использования по своему содержанию является выполнением последовательности действий, которая инициализируется посредством экземпляра сообщения от экземпляра актера. В качестве ответной реакции на сообщение актера выполняется последовательность действий, установленная для данного варианта использования. При этом актеры могут генерировать новые сообщения для инициирования вариантов использования.

Подобное взаимодействие будет продолжаться до тех пор, пока не закончится выполнение требуемой последовательности действий экземпляром варианта использования и указанный в модели экземпляр актера (и никакой другой) не получит требуемый экземпляр сервиса. Окончание взаимодействия означает отсутствие инициализации сообщений от актеров для базовых вариантов использования.

Варианты использования могут быть специфицированы в виде текста, которые в последующем могут стать прототипами операций и методов совместно с атрибутами, в виде графа деятельности, посредством конечного автомата или любого другого механизма описания поведения, включающего предусловия и постусловия. Взаимодействие между вариантами использования и актерами может уточняться на диаграмме кооперации, когда описываются взаимосвязи между системой, содержащей эти варианты использования, и окружением или внешней средой этой системы.

В случае, когда для представления иерархической структуры проектируемой системы используются подсистемы, система может быть определена в виде вариантов использования на всех уровнях. Отдельные подсистемы или классы могут реализовывать такие варианты использования. При этом наиболее общий или абстрактный вариант использования в последующем может уточняться множеством более частных вариантов использования, каждый из которых будет определять сервис элемента модели, содержащийся в сервисе исходной системы. В этом контексте общий вариант использования может рассматриваться как суперсервис для уточняющих его подвариантов, которые, в свою очередь, могут рассматриваться как подсервисы исходного варианта использования. Функциональность, определенная для более общего варианта использования, полностью наследуется всеми вариантами нижних уровней.

Однако следует заметить, что для представления структуры подсистемы недостаточно вариантов использования, поскольку они могут представлять только функциональность отдельных элементов модели. Подчиненные варианты использования могут кооперироваться для совместного выполнения суперсервиса варианта использования верхнего уровня. Эта кооперация также может быть представлена на диаграмме кооперации в виде совместных действий отдельных элементов модели.

Отдельные варианты использования нижнего уровня могут участвовать в нескольких кооперациях, т. е. играть определенную роль при выполнении сервисов нескольких вариантов верхнего уровня. Для отдельных таких кооперацій могут быть определены соответствующие роли актеров, взаимодействующих с конкретными вариантами использования нижнего уровня. Эти роли будут играть актеры нижнего уровня модели системы. Хотя некоторые из таких актеров могут быть актерами верхнего уровня, это не противоречит принятым в языке UML семантическим правилам построения диаграмм вариантов использования.

Окружение вариантов использования нижнего уровня является самостоятельным элементом модели, который, в свою очередь, может содержать другие элементы модели, определенные для этих вариантов использования. Таким образом, с точки зрения общего представления концептуальной модели, взаимодействие между вариантами использования нижнего уровня определяет результат выполнения сервиса варианта верхнего уровня.

Реализация варианта использования зависит от типа элемента модели, в котором он определен. Например, варианты использования моделируемой программной системы могут быть реализованы посредством операций классов модели. Применительно к бизнес-системам варианты использования могут реализоваться сотрудниками этой системы. Во всех случаях элементы системы должны взаимодействовать друг с другом для совместного обеспечения требуемого поведения и выполнения вариантов использования модели.

Совместное обеспечение требуемого поведения описывается специальным элементом языка UML — кооперация или сотрудничество, который будет рассмотрен в *главе 6*, посвященной построению диаграмм кооперации. Здесь лишь отметим, что кооперации используются как для уточнения спецификаций в виде вариантов использования нижних уровней диаграммы, так и для описания особенностей их последующей реализации.

Если в качестве моделируемой сущности выступает подсистема нижнего уровня, то отдельные пользователи вариантов использования этой подсистемы моделируются актерами. Такие актеры могут являться внутренними сотрудниками по отношению к моделируемым системам верхних уровней, что зачастую в явном виде не указываются на диаграммах. Эти элементы модели могут содержаться в других пакетах или подсистемах. В последнем случае роли актеров определяются в том пакете, к которому относится соответствующая подсистема.

С системно-аналитической точки зрения построение диаграммы вариантов использования специфицирует не только функциональные требования к проектируемой системе, но и выполняет исходную структуризацию предметной области. Последняя задача сочетает в себе не только следование рекомендациям ООАП, но и является в некотором роде искусством, умением выделять главное в модели системы. Хотя рациональный унифицированный

процесс не исключает итеративный возврат к диаграмме вариантов использования для ее модификации на последующих этапах работы над проектом, не вызывает сомнений тот факт, что любая подобная модификация потребует, как по цепочке, изменений во всех других представлениях и диаграммах системы. Поэтому всегда необходимо стремиться к возможно более точному представлению модели именно в форме диаграммы вариантов использования.

Если же варианты использования применяются для спецификации части системы, то эти варианты будут эквивалентны соответствующим вариантам использования в модели подсистемы соответствующего пакета. Важно понимать, что все сервисы системы должны быть явно определены на диаграмме вариантов использования, и никаких других сервисов, которые отсутствуют на данной диаграмме, проектируемая система не может выполнять по определению. Более того, если для моделирования поведения системы используются сразу несколько моделей (например, модель анализа и модель проектирования), то множество вариантов использования всех пакетов системы должно быть эквивалентно множеству вариантов использования модели в целом.

Ниже приводится перечень типичных ошибок, которые встречаются при разработке диаграмм вариантов использования.

1. Излишняя детализация диаграммы вариантов использования и ее неявное превращение в диаграмму деятельности за счет желания отразить все детали и логику поведения моделируемой системы.
2. Вместо текста сценария варианта использования записываются функциональные требования к системе без указания последовательности выполняемых системой действий.
3. Разработчики приступают к спецификации атрибутов и операций классов до того, как сформулированы и написаны сценарии всех вариантов использования модели.
4. В качестве инициатора действий на диаграмме или в сценарии выступает проектируемая система.
5. Вариантам использования даются слишком краткие имена или аббревиатуры, смысл которых понятен узкому кругу посвященных.
6. Логическая привязка модели к существующим техническим устройствам или способам взаимодействия вариантов использования и актеров.
7. Написание текста сценария вариантов использования в терминологии, непонятной пользователям системы и заказчику. В модели описываются только действий актеров и игнорируются действия моделируемой системы.
8. В тексте сценария отсутствуют описания альтернативных последовательностей действий или исключений, которые могут быть разработчиками просто упущены из внимания.

9. Тратится слишком много времени на графическое совершенство диаграммы вариантов использования и решение вопросов об именах отдельных элементов диаграммы.

В заключение приводится сводка всех рассмотренных графических обозначений, которые могут встретиться при построении диаграмм вариантов использования (табл. 4.5). Этого перечня может оказаться вполне достаточно для выполнения большинства реальных проектов.

Таблица 4.5. *Графические элементы диаграмм вариантов использования*

Графическое изображение	Название
	Вариант использования (use case)
	Актер (actor)
	Примечание (note)
	Ассоциация (association relationship)
<<include>>	Включение (include relationship)
<<extend>>	Расширение (extend relationship)
	Обобщение (generalization relationship)

Таблица 4.5 *(окончание)*

Графическое изображение	Название
	Бизнес-актер (business actor)
	Сотрудник (business worker)
	Бизнес-вариант использования (business use case)

Глава 5

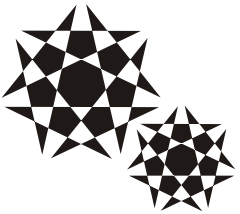


Диаграмма классов (class diagram)

Центральное место в методологии ООАП занимает разработка логической модели системы в виде диаграммы классов. Нотация классов в языке UML проста и интуитивно понятна всем тем, кто когда-либо имел опыт работы с CASE-средствами. Схожая нотация применяется для объектов, которые в контексте языка UML являются экземплярами классов и изображаются на канонических диаграммах различных типов.

Нотация UML предоставляет широкие возможности для отображения дополнительной информации (абстрактные операции и классы, стереотипы, общие и частные методы, детализированные интерфейсы, параметризованные классы). При этом могут использоваться графические изображения для ассоциаций и их специфических свойств, таких как отношение агрегации, когда составными частями класса могут выступать другие классы.

Диаграмма классов (class diagram) служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования. Диаграмма классов может отражать, в частности, различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений. На данной диаграмме не указывается информация о временных аспектах функционирования системы. С этой точки зрения диаграмма классов может служить дальнейшим развитием концептуальной модели проектируемой системы.

В общем случае диаграмма классов представляет собой конечный граф, вершинами которого являются элементы типа "классификатор", которые связаны различными типами структурных отношений. Следует заметить, что диаграмма классов может также содержать интерфейсы, пакеты, отношения и даже отдельные экземпляры классификаторов, такие как объекты и связи. Когда говорят о данной диаграмме, то имеют в виду статическую структурную модель проектируемой системы. Поэтому диаграмму классов принято считать графическим представлением таких структурных взаимосвязей логической модели системы, которые не зависят от времени.

Диаграмма классов состоит из множества элементов, которые в совокупности отражают декларативные знания о предметной области. Эти знания интерпретируются в базовых понятиях языка UML, таких как классы, интерфейсы и отношения между ними и составляющими их элементами. При этом отдельные элементы этой диаграммы могут организовываться в пакеты для представления более общей модели системы. Если же диаграмма классов является частью некоторого пакета, то ее элементы должны соответствовать элементам этого пакета, включая возможные ссылки на элементы из других пакетов.

В общем случае пакет статической структурной модели может быть представлен в виде одной или нескольких диаграмм классов. Декомпозиция подобного представления на отдельные диаграммы выполняется с целью удобства и графической визуализации структурных взаимосвязей предметной области. При этом элементы диаграммы классов соответствуют элементам статической семантической модели. Модель системы, в свою очередь, должна быть согласована с внутренней структурой классов, которая описывается на языке UML.

5.1. Класс

Класс (class) в языке UML является абстрактным описанием или представлением свойств множества объектов, которые обладают одинаковой структурой, поведением и отношениями с объектами из других классов. Графически класс в нотации языка UML изображается в виде прямоугольника, который дополнительно может быть разделен горизонтальными линиями на разделы или секции (рис. 5.1). В этих секциях могут указываться имя класса, атрибуты (переменные) и операции (методы).

Рис. 5.1. Варианты графического изображения классов на диаграмме классов

Обязательным элементом обозначения класса является его имя. На начальных этапах разработки диаграммы отдельные классы могут обозначаться простым прямоугольником с указанием только имени соответствующего класса (рис. 5.1, *а*). По мере проработки отдельных компонентов диаграммы описания классов дополняются атрибутами (рис. 5.1, *б*) и операциями

(рис. 5.1, *в*). Предполагается, что окончательный вариант диаграммы содержит наиболее полное описание классов, которые состоят из трех разделов или секций.

Даже если секция атрибутов и операций является пустой, в обозначении класса она выделяется горизонтальной линией, чтобы сразу отличить класс от других элементов языка UML. Примеры графического изображения конкретных классов приведены на рис. 5.2. В первом случае для класса `Прямоугольник` (рис. 5.2, *а*) указаны только его атрибуты — точки на координатной плоскости, которые определяют его расположение. Для класса `Окно` (рис. 5.2, *б*) указаны только его операции, при этом секция его атрибутов оставлена пустой.

Иногда в обозначениях классов используется дополнительный четвертый раздел, в котором приводится информация справочного характера (например, сведения о разработчике, языке реализации и пр.) или явно указываются исключительные ситуации. Для класса `Счет` (рис. 5.2, *в*) дополнительно изображена четвертая секция, в которой указано исключение — неверно введенный ПИН-код.

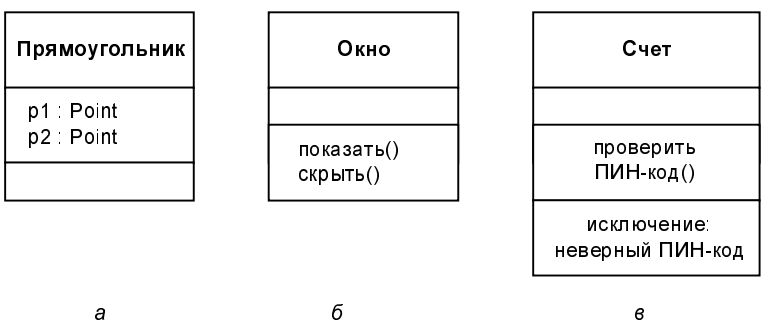


Рис. 5.2. Примеры графического изображения конкретных классов

5.1.1. Имя класса

Имя класса должно быть уникальным в пределах пакета, который может содержать несколько диаграмм классов или, возможно, только одну диаграмму. Имя указывается в самой верхней секции прямоугольника, поэтому эта секция часто называется секцией имени класса. В дополнение к общему правилу именования элементов языка UML, имя класса записывается по центру секции имени полужирным шрифтом и должно начинаться с заглавной буквы. Рекомендуется в качестве имен классов использовать существительные, записанные по практическим соображениям без пробелов. Необходимо помнить, что именно имена классов образуют словарь предметной области при ООАП.

Примечание

Хотя согласно требованиям спецификации языка UML имя класса должно записываться полужирным шрифтом, это условие соблюдается разработчиками крайне редко. Возможно, причиной служит тот факт, что CASE-средства не поддерживают соответствующую нотацию. В дальнейшем толщине шрифта при записи имени класса не будет придаваться принципиального значения, поскольку на фоне других ошибок, допускаемых разработчиками при графическом изображении элементов языка UML, эта представляется легким недоразумением.

В секции имени класса могут также находиться стереотипы или ссылки на стандартные шаблоны, от которых образован данный класс и, соответственно, от которых он наследует атрибуты и операции. В этой секции может также приводиться информация о разработчике данного класса и статус состояния разработки, а также могут записываться и другие общие свойства этого класса, имеющие отношение к другим классам диаграммы или стандартным элементам языка UML.

Примерами имен классов могут быть такие существительные, как Сотрудник, Компания, Руководитель, Клиент, Продавец, Менеджер, Офис и многие другие, имеющие непосредственное отношение к моделируемой предметной области и функциональному назначению проектируемой системы.

Класс может не иметь экземпляров или объектов. В этом случае он называется *абстрактным* классом, а для обозначения его имени используется наклонный шрифт (курсив). В языке UML принято общее соглашение о том, что любой текст, относящийся к абстрактному элементу, записывается курсивом.

Примечание

Курсив при записи имени класса, в отличие от толщины шрифта, имеет принципиальное значение, поскольку является семантическим аспектом описания абстрактных элементов языка UML. Именно по этой причине разработчикам следует внимательно записывать имена классов.

В некоторых случаях необходимо явно указать, к какому пакету относится тот или иной класс. Для этой цели используется специальный символ разделитель — двойное двоеточие (::). Синтаксис строки имени класса в этом случае будет следующий: `<Имя пакета>::<Имя класса>`. Другими словами, перед именем класса должно быть явно указано имя пакета, к которому его следует отнести. Например, если определен пакет с именем Банк, то класс `Счет` в этом банке может быть записан в виде: `Банк::Счет`.

5.1.2. Атрибуты класса

Атрибут (attribute) класса служит для представления отдельного свойства или признака, который является общим для всех объектов данного класса. Атрибуты класса записываются во второй сверху секции прямоугольника класса, поэтому эту секцию часто называют *секцией атрибутов*.

В языке UML принята определенная стандартизация записи атрибутов класса, которая подчиняется некоторым синтаксическим правилам. Каждому атрибуту класса соответствует отдельная строка текста, которая состоит из квантора видимости, имени, его кратности, типа значений атрибута и, возможно, его исходного значения. Общий формат записи отдельного атрибута класса следующий:

```
<квантор видимости> <имя атрибута> [кратность]:
<тип атрибута> =  <исходное значение> {строка-свойство}
```

Квантор видимости (visibility) может принимать одно из 4-х возможных значений и, соответственно, отображается при помощи специальных символов.

- Символ "+" — обозначает атрибут с областью видимости типа общедоступный (public). Атрибут с этой областью видимости доступен или виден из любого другого класса пакета, в котором определена диаграмма.
- Символ "#" — обозначает атрибут с областью видимости типа защищенный (protected). Атрибут с этой областью видимости недоступен или невиден для всех классов, за исключением подклассов данного класса.
- Символ "-" — обозначает атрибут с областью видимости типа закрытый (private). Атрибут с этой областью видимости недоступен или невиден для всех классов без исключения.
- И, наконец, символ "~" обозначает атрибут с областью видимости типа пакетный (package). Атрибут с этой областью видимости недоступен или невиден для всех классов за пределами пакета, в котором определен класс-владелец данного атрибута.

Квантор видимости может быть опущен. В этом случае его отсутствие просто означает, что видимость атрибута не указывается. Эта ситуация отличается от принятых по умолчанию соглашений в традиционных языках программирования, когда отсутствие квантора видимости трактуется как public или private. Однако вместо условных графических обозначений можно записывать соответствующее ключевое слово: public, protected, private или package.

Примечание

По справедливому замечанию Мартина Фаулера — ничто не определяется так просто и не вызывает столько противоречивых толкований в языке UML, как интерпретация видимости на диаграммах. Хотя язык UML по определению инвариантен относительно реализации своих конструкций в конкретных языках программирования, попытка интерпретировать кванторы видимости безотносительно к языкам программирования может привести к серьезным трудностям. Именно по этой причине значения кванторов видимости атрибутов и операций рекомендуется записывать в модели только после того, как будет принято решение о языке ее программной реализации. С другой стороны, разработчик всегда может уточнить семантику используемых кванторов видимости в форме примечания с пояснительным текстом на естественном языке.

Имя атрибута представляет собой строку текста, которая используется в качестве идентификатора соответствующего атрибута и поэтому должна быть уникальной в пределах данного класса. Имя атрибута является единственным обязательным элементом синтаксического обозначения атрибута, должно начинаться со срочной (малой) буквы и, как правило, не должно содержать пробелов.

Кратность атрибута характеризует общее количество конкретных атрибутов данного типа, входящих в состав отдельного класса. В общем случае кратность записывается в форме строки из цифр в квадратных скобках после имени соответствующего атрибута, при этом цифры разделяются двоеточием:

```
[нижняя граница .. верхняя граница],
```

где *нижняя граница* и *верхняя граница* являются положительными целыми числами, каждая пара которых служит для обозначения отдельного замкнутого интервала целых чисел. В качестве *верхней границы* может использоваться специальный символ "*" (звездочка), который означает произвольное положительное целое число. Другими словами, этот символ означает неограниченное сверху значение кратности соответствующего атрибута.

Интервалов кратности для отдельного атрибута может быть несколько. В этом случае их совместное использование соответствует теоретико-множественному объединению соответствующих интервалов. Значения кратности из интервала следуют в монотонно возрастающем порядке без пропуска отдельных чисел, лежащих между нижней и верхней границами. При этом придерживаются следующего правила: соответствующие нижние и верхние границы интервалов включаются в значение кратности.

Если в качестве кратности указывается единственное число, то кратность атрибута принимается равной данному числу. Если же указывается единственный знак "*", то это означает, что кратность атрибута может быть произвольным положительным целым числом или нулем.

В качестве примера рассмотрим следующие варианты задания кратности атрибутов.

- `[0..1]` — означает, что кратность атрибута может принимать значение 0 или 1. При этом 0 означает отсутствие данного атрибута у отдельных объектов рассматриваемого класса.
- `[0..*]` — означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 0. Эта кратность может быть записана короче в виде простого символа — `[*]`.
- `[1..*]` — означает, что кратность атрибута может принимать любое положительное целое значение большее или равное 1.
- `[1..5]` — означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 4, 5.

- `[1..3,5,7]` — означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 5, 7.
- `[1..3,7..10]` — означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, 7, 8, 9, 10.
- `[1..3,7..*]` — означает, что кратность атрибута может принимать любое значение из чисел: 1, 2, 3, а также любое положительное целое значение большее или равное 7.

Если кратность атрибута не указана, то по умолчанию принимается ее значение равное `[1..1]`, т. е. в точности 1.

Тип атрибута представляет собой выражение, семантика которого определяется некоторым типом данных, определенным в пакете Типы данных языка UML или разработчиком. В нотации UML тип атрибута иногда определяется в зависимости от языка программирования, который предполагается использовать для реализации данной модели. В простейшем случае тип атрибута указывается строкой текста, имеющей осмысленное значение в пределах пакета или модели, к которым относится рассматриваемый класс. Типу атрибута должно предшествовать двоеточие.

Можно привести следующие примеры задания имен и типов атрибутов классов:

- `цвет : Color` — здесь `цвет` является именем атрибута, `Color` — именем типа данного атрибута. Указанная запись может определять традиционно используемую RGB-модель (красный, зеленый, синий) для представления цвета. В этом случае имя типа `Color` как раз и характеризует семантическую конструкцию, которая применяется в большинстве языков программирования для представления цвета;
- `имяСотрудника[1..2] : String` — здесь `имяСотрудника` является именем атрибута, который служит для представления информации об имени, а возможно и отчестве конкретного сотрудника. Тип атрибута `String` (Строка) как раз и указывает на тот факт, что отдельное значение имени представляет собой строку текста из одного или двух слов (например, "Кирилл" или "Дмитрий Иванович");

Примечание

Поскольку во многих языках программирования существует тип данных String, использование соответствующего англоязычного термина не вызывает недоразумения у большинства программистов. Поскольку в языке UML все термины даются в англоязычном представлении, использование в качестве типа атрибута Строка в данной ситуации не исключается и определяется только соображениями удобства.

- `видимость : Boolean` — в данном случае `видимость` является именем атрибута, который может характеризовать наличие визуального представления

соответствующего класса на экране монитора. В этом случае тип Boolean означает, что возможными значениями данного атрибута являются одно из двух логических значений: *истина* (true) или *ложь* (false). При этом значение "истина" может соответствовать наличию графического изображения на экране монитора, а значение "ложь" — его отсутствию, о чем дополнительно указывается в пояснительном тексте;

- `Форма : Многоугольник` — здесь имя атрибута `форма` может характеризовать класс, который является геометрической фигурой на плоскости. В этом случае тип атрибута `Многоугольник` указывает на тот факт, что отдельная геометрическая фигура может иметь форму треугольника, прямоугольника, ромба, пятиугольника и любого другого многоугольника, но не окружности или эллипса.

Исходное значение служит для задания некоторого начального значения для соответствующего атрибута в момент создания отдельного экземпляра класса. Здесь необходимо придерживаться правила принадлежности значения типу конкретного атрибута. Если исходное значение не указано, то значение соответствующего атрибута не определено на момент создания нового экземпляра класса. С другой стороны, конструктор соответствующего объекта может переопределять исходное значение в процессе выполнения программы, если в этом возникает необходимость.

В качестве примеров исходных значений атрибутов можно привести дополненные варианты задания атрибутов.

- `цвет : Color = (255,0,0)` — в RGB-модели цвета это соответствует чистому красному цвету в качестве исходного значения для данного атрибута.
- `имяСотрудника[1..2] : String = 'Иван Иванович'` — возможно, это нетипичный случай, который, скорее, соответствует ситуации `имяРуководителя[2]:String = 'Иван Иванович'`.
- `видимость : Boolean = истина` — может соответствовать ситуации, когда в момент создания экземпляра класса создается видимое на экране монитора окно, соответствующее данному объекту.
- `форма : Многоугольник = прямоугольник` — вряд ли требует комментариев, поскольку здесь речь идет о геометрической форме создаваемого объекта.

При задании атрибутов могут быть использованы две дополнительные синтаксические конструкции — это подчеркивание строки атрибута и пояснительный текст в фигурных скобках.

Подчеркивание строки атрибута означает, что соответствующий атрибут является общим для всех объектов данного класса, т. е. его значение у всех создаваемых объектов одинаковое (аналог ключевого слова `static` в некоторых языках программирования). Например, если некоторый атрибут задан в виде `форма : Прямоугольник`, то это будет означать, что все объекты данного класса имеют форму прямоугольника.

Строка-свойство служит для указания дополнительных свойств атрибута, которые могут характеризовать особенности изменения значений атрибута в ходе выполнения соответствующей программы. Фигурные скобки как раз и обозначают фиксированное значение соответствующего атрибута для класса в целом, которое должны принимать все вновь создаваемые экземпляры класса без исключения. Это значение принимается за исходное значение атрибута, которое не может быть переопределено в последующем. Отсутствие строки-свойства по умолчанию трактуется так, что значение соответствующего атрибута может быть изменено в программе.

Например, строка-свойство, в записи атрибута, `заработнаяПлата : Currency = $500{frozen}` может служить для обозначения фиксированной заработной платы для каждого объекта класса `Сотрудник` в некоторой организации, которое не подлежит изменению с течением времени. Именно на этот факт указывает строка-свойство `{frozen}` (замороженный), что, вообще говоря, не придаст оптимизма в работе этих сотрудников.

С другой стороны, запись данного атрибута без строки-свойства, в виде `заработнаяПлата : Currency = $500`, означает уже нечто иное, а именно: при создании нового экземпляра класса `Сотрудник` (пример — прием на работу нового сотрудника) для него по умолчанию устанавливается заработная плата в $500. Однако для отдельных сотрудников могут быть сделаны исключения как в большую, так и в меньшую сторону, и это значение со временем может быть изменено.

5.1.3. Операции класса

Операция (operation) — это некоторый сервис, который предоставляет каждый экземпляр или объект класса по требованию своих клиентов (других объектов, в том числе и экземпляров данного класса). Операции класса записываются в третьей сверху секции прямоугольника класса, поэтому эту секцию часто называют *секцией операций*. Совокупность операций характеризует функциональный аспект поведения всех объектов данного класса.

Запись операций класса в языке UML также стандартизована и подчиняется определенным синтаксическим правилам. При этом каждой операции класса соответствует отдельная строка, которая состоит из квантора видимости операции, имени, выражения типа возвращаемого значения, и, возможно, строка-свойство данной операции. Общий формат записи отдельной операции класса следующий:

```
<квантор видимости> <имя операции>(список параметров):
<выражение типа возвращаемого значения>
{строка-свойство}
```

Квантор видимости, как и в случае атрибутов класса, может принимать одно из четырех возможных значений и, соответственно, отображается при

помощи специального символа либо ключевого слова. Символ "+" обозначает операцию с областью видимости типа общедоступный (public). Символ "#" обозначает операцию с областью видимости типа защищенный (protected). Символ "-" используется для обозначения операции с областью видимости типа закрытый (private). И, наконец, символ "~" используется для обозначения операции с областью видимости типа пакетный (package).

Квантор видимости для операции может быть опущен. В этом случае его отсутствие просто означает, что видимость операции не указывается. Вместо условных графических обозначений также можно записывать соответствующее ключевое слово: public, protected, private или package.

Примечание

Применительно к конкретным языкам программирования могут быть определены дополнительные кванторы видимости, как, например, видимость friend (дружественный) в языке C++, которая никак не специфицирована в языке UML. В этом случае подобные дополнения являются расширением базовой нотации и требуют от разработчика соответствующих пояснений в форме текста на естественном языке или в виде строки-свойства.

Имя операции представляет собой строку текста, которая используется в качестве идентификатора соответствующей операции и поэтому должна быть уникальной в пределах данного класса. Имя операции является единственным обязательным элементом синтаксического обозначения операции, должно начинаться со срочной (малой) буквы и, как правило, не должно содержать пробелов.

Список параметров является перечнем разделенных запятой формальных параметров, каждый из которых, в свою очередь, может быть представлен в следующем виде:

```
<вид параметра> <имя параметра> : <выражение типа> =
<значение параметра по умолчанию>
```

Здесь *вид параметра* — есть одно из ключевых слов `in`, `out` или `inout` со значением `in` по умолчанию, в случае если он не указывается. *Имя параметра* есть идентификатор соответствующего формального параметра, при записи которого следуют правилам задания имен атрибутов. *Выражение типа* является спецификацией типа данных для возвращаемого значения соответствующего формального параметра. Наконец, *значение по умолчанию* в общем случае представляет собой некоторое конкретное значение для этого формального параметра.

Выражение типа возвращаемого значения также указывает на тип данных значения, которое возвращается объектом после выполнения соответствующей операции. Двоеточие и выражение типа возвращаемого значения могут быть опущены, если операция не возвращает никакого значения. Для указания нескольких возвращаемых значений данный элемент спецификации операции может быть записан в виде списка отдельных выражений.

Операция с областью действия на весь класс показывается подчеркиванием имени и строки выражения типа. В этом случае под областью действия операции понимаются все объекты этого класса. В этом случае вся строка записи операции подчеркивается.

Строка-свойство служит для указания значений свойств, которые могут быть применены к данной операции. Строка-свойство не является обязательной, она может отсутствовать, если никакие свойства не специфицированы.

Например, если операция не должна изменять состояние системы и, соответственно, не имеет никакого побочного эффекта, то в ее записи следует указать строку-свойство `{query}` (запрос). В противном случае операция может изменять состояние системы, хотя нет никаких гарантий, что она будет это делать.

Для повышения производительности системы одни операции могут выполняться параллельно, а другие — только последовательно. В этом случае для указания параллельности выполнения операции также можно использовать строку-свойство вида `{concurrency = имя}`, где *имя* может принимать одно из следующих значений: `sequential` (последовательная), `concurrent` (параллельная), `guarded` (охраняемая). При этом придерживаются следующей семантики для данных значений:

- `sequential` (последовательная) — для данной операции необходимо обеспечить ее единственное выполнение в системе, одновременное выполнение других операций может привести к ошибкам или нарушениям целостности объектов класса;
- `concurrent` (параллельная) — данная операция в силу своих особенностей может выполняться параллельно с другими операциями в системе, при этом параллельность должна поддерживаться на уровне реализации модели;
- `guarded` (охраняемая) — все обращения к данной операции должны быть строго упорядочены во времени с целью сохранения целостности объектов данного класса, при этом могут быть приняты дополнительные меры по контролю исключительных ситуаций на этапе ее выполнения.

Примечание

С целью сокращения обозначений допускается использование одного имени в качестве строки-свойства для указания соответствующего значения параллельности. Отсутствие данной строки-свойства означает, что семантика параллельности для операции не определена. Поэтому следует предположить худший с точки зрения производительности случай, когда данная операция требует последовательного выполнения.

Появление сигнатуры операции у классов на самом верхнем уровне абстракции объявляет данную операцию на весь класс, при этом данная операция

наследуется всеми потомками данного класса. Если в некотором классе операция не реализуется (т. е. объекты данного класса не могут выполнять эту операцию), то такая операция называется *абстрактной* и может быть помечена строкой-свойством `{abstract}`. Другой способ показать абстрактный характер операции — записать ее сигнатуру курсивом. Запись данной операции без свойства `{abstract}` у классов-потомков может указывать на тот факт, что соответствующие объекты данного класса-потомка могут выполнять данную операцию в качестве своего метода.

Примечание

Если для некоторой операции необходимо дополнительно указать особенности ее реализации (например, алгоритм), то на диаграмме классов это может быть сделано только в форме примечания, записанного в виде текста, который присоединяется к записи операции в соответствующей секции класса. Для представления собственно алгоритмов реализации отдельных операций служит другой тип канонических диаграмм — диаграмма деятельности.

Если объекты класса принимают и реагируют на некоторый сигнал, то запись данной операции может быть помечена стереотипом <<signal>> (сигнал). Это обозначение равнозначно обозначению некоторой операции. Реакция объекта на прием сигнала может быть показана в виде некоторого конечного автомата (диаграммы состояний). Эта нотация может быть использована, чтобы показать реакцию объектов класса на некоторые ошибочные ситуации или исключения, которые могут моделироваться как сигналы или сообщения.

Поведение операции может быть дополнительно специфицировано в форме присоединенного к операции примечания. Если текст примечания представляет собой формальную спецификацию на некотором языке программирования, то в этом случае он заключается в скобки, что соответствует элементу "семантическое ограничение языка UML". В противном случае текст примечания является описанием на естественном языке, а само примечание обозначается прямоугольником с "загнутым" верхним правым уголком *(см. главу 4)*.

Список формальных параметров и тип возвращаемого значения могут не указываться. Квантор видимости атрибутов и операций может быть указан в виде специального значка или символа, которые используются для графического представления моделей в некотором инструментальном средстве. Еще раз следует напомнить, что имена операций, так же как и атрибутов, записываются со строчной (малой) буквы, а их типы параметров — с заглавной (большой) буквы. При этом обязательной частью строки записи операции является наличие имени операции и круглых скобок.

В качестве примеров записи операций можно привести несколько обозначений отдельных операций.

- `+нарисовать(форма : Многоугольник = прямоугольник, цветЗаливки : Color (0, 0, 255))` — обозначает общедоступную операцию по изобра-

жению на экране монитора прямоугольной области синего цвета, если не указываются другие значения в качестве аргументов данной операции.

- `-изменитьСчетКлиента(номерСчета : Integer): Currency` — обозначает закрытую операцию по изменению средств на текущем счете клиента банка. При этом аргументом данной операции является номер счета клиента, который записывается в виде целого числа (например, "123456"). Результатом выполнения этой операции является некоторое число, записанное в принятом денежном формате (например, $1,500.00).
- `#выдатьСообщение( ):{'Ошибка деления на ноль'}` — смысл данной защищенной операции не требует пояснения, поскольку содержится в строке-свойстве операции. Данное сообщение может появиться на экране монитора в случае попытки деления некоторого числа на ноль, что недопустимо.
- *`создать()`* — может обозначать абстрактную операцию по созданию отдельного объекта класса (конструктор), которая не содержит формальных параметров. Эта операция не может быть вызвана объектами класса, в котором она определена (если этот класс, в свою очередь, является абстрактным, то на его основе не могут быть созданы и объекты). Для использования данной операции ее необходимо специфицировать в классах-потомках.

5.2. Отношения между классами

Кроме внутреннего устройства или структуры классов важную роль при разработке проектируемой системы имеют различные отношения между классами, которые также могут быть изображены на диаграмме классов. Однако совокупность допустимых типов таких отношений строго фиксирована в языке UML и определяется самой семантикой этих отношений. Базовыми отношениями или взаимосвязями, которые могут изображаться на диаграммах классов, являются:

- отношение ассоциации (association relationship);
- отношение обобщения (generalization relationship);
- отношение агрегации (aggregation relationship);
- отношение композиции (composition relationship);
- отношение зависимости (dependency relationship).

Каждое из этих отношений имеет собственное графическое представление, которое отражает характер взаимосвязи между объектами соответствующих классов.

5.2.1. Отношение ассоциации

Отношение ассоциации соответствует наличию произвольного отношения или взаимосвязи между классами. Данное отношение, как уже отмечалось в *главе 4*, обозначается сплошной линией со стрелкой или без нее, и с некоторыми дополнительными символами, которые характеризуют специальные свойства ассоциации. Хотя ассоциации и рассматривались при изучении элементов диаграммы вариантов использования, применительно к диаграммам классов семантика этого типа отношений значительно расширена. В качестве дополнительных специальных символов могут использоваться имя ассоциации, символ навигации, а также имена и кратность классов-ролей ассоциации.

Имя ассоциации является необязательным элементом ее обозначения. Однако если оно задано, то записывается с заглавной (большой) буквы рядом с линией соответствующей ассоциации. Отдельные классы ассоциации могут играть некоторую *роль* в соответствующем отношении, что может быть явно указано на диаграмме заданием имени концевых точек ассоциации.

Наиболее простой случай данного отношения — *бинарная ассоциация*. Она связывает в точности два различных класса и может быть ненаправленным (симметричным) или направленным отношением. Частным случаем бинарной ассоциации является *рефлексивная* ассоциация, которая связывает класс с самим собой.

Ненаправленная бинарная ассоциация изображается линией без стрелки. Для нее на диаграмме может быть указан порядок чтения классов с использованием значка в форме треугольника рядом с именем данной ассоциации.

В качестве простого примера ненаправленной бинарной ассоциации рассмотрим отношение между двумя классами — классом `Компания` и классом `Сотрудник` (рис. 5.3). Они связаны между собой бинарной ассоциацией `Работает`, имя которой указано на рисунке рядом с линией ассоциации. Для данного отношения определен порядок чтения следования классов, `Сотрудник Работает в Компании`. Отдельным примером или экземпляром данного отношения может являться пара значений (`Петров И.И.`, "`Рога & Копыта`"). Наличие данной ассоциации означает, что сотрудник Петров И. И. работает в компании "Рога & Копыта".

Направленная бинарная ассоциация изображается сплошной линией с простой стрелкой на одном из ее концевых точек. Направление этой стрелки указывает на то, какой из классов является первым, а какой — вторым (на него указывает стрелка ассоциации).

В качестве простого примера направленной бинарной ассоциации рассмотрим отношение между двумя классами — классом `Многоугольник` и классом `Сторона` (рис. 5.4). Они связаны между собой бинарной ассоциацией `Содержит`, для которой определен порядок следования классов. Это означает, что

конкретный объект класса `Многоугольник` всегда должен указываться первым при рассмотрении взаимосвязи с объектом класса `Сторона`. Другими словами, эти соответствующие объекты классов образуют кортеж элементов, например, <треугольник, сторона1, сторона2, сторона3>.

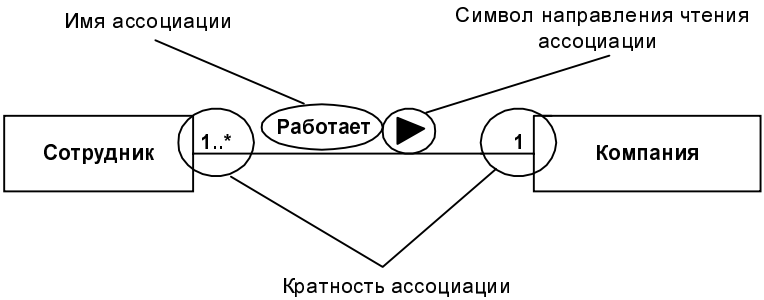


Рис. 5.3. Графическое изображение ненаправленной бинарной ассоциации между классами

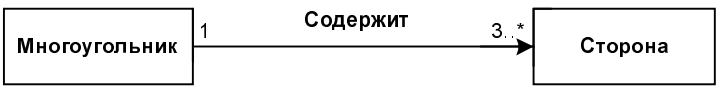


Рис. 5.4. Графическое изображение направленной бинарной ассоциации между классами

Частным случаем отношения ассоциации является так называемая *исключающая ассоциация* (Xor-association). Семантика данной ассоциации указывает на тот факт, что из нескольких потенциально возможных вариантов данной ассоциации в каждый момент может использоваться только один ее экземпляр. На диаграмме классов исключающая ассоциация изображается пунктирной линией, соединяющей две и более ассоциации, рядом с которой записывается строка-ограничение `{xor}`.

Например, счет в банке может быть открыт для клиента, в качестве которого может выступать либо физическое лицо (индивидуум), либо юридическое лицо (компания), что может быть изображено с помощью исключающей ассоциации (рис. 5.5).

Более сложный случай представляет собой *тернарная ассоциация*, которая связывает некоторым отношением три класса. Ассоциация более высокой арности в общем случае называется *N-арной ассоциацией* (читается — "эн арная ассоциация"). Такая ассоциация связывает некоторым отношением более трех классов, при этом один класс может участвовать в ассоциации более чем один раз. Каждый экземпляр *N*-арной ассоциации представляет собой *N*-арный кортеж, состоящий из объектов соответствующих классов.

В этом контексте бинарная ассоциация является частным случаем N-арной ассоциации, когда значение $N = 2$, но имеет свое собственное обозначение.

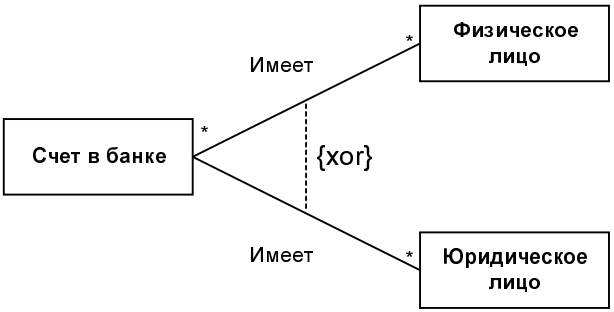


Рис. 5.5. Графическое изображение исключающей ассоциации между тремя классами

В общем случае N-арная ассоциация графически обозначается *ромбом*, от которого ведут линии к символам классов данной ассоциации. Сам же ромб соединяется с символами соответствующих классов сплошными линиями. Обычно линии проводятся от вершин или от середины сторон. Имя N-арной ассоциации записывается рядом с ромбом соответствующей ассоциации. Однако порядок классов в N-арной ассоциации, в отличие от порядка множеств в отношении, на диаграмме не фиксируется.

В качестве примера конкретной тернарной ассоциации рассмотрим отношение между тремя классами: `Футбольная команда`, `Год` и `Игра`. Данная ассоциация указывает на наличие отношения между этими тремя классами, которое может представлять информацию об играх конкретных пар футбольных команд в национальном чемпионате в течение нескольких последних лет (рис. 5.6).

Рис. 5.6. Графическое изображение тернарной ассоциации между тремя классами

Некоторый класс может быть присоединен к линии ассоциации пунктирной линией. Это означает, что данный класс обеспечивает поддержку свойств

соответствующей N-арной ассоциации, а сама N-арная ассоциация имеет атрибуты, операции и/или ассоциации. Другими словами, такая ассоциация является классом с соответствующим обозначением в виде прямоугольника и является самостоятельным элементом языка UML — *ассоциацией-классом* (Association Class).

Как уже упоминалось, отдельный класс в ассоциации может играть определенную роль в данной ассоциации. Эта роль может быть явно специфицирована на диаграмме классов. С этой целью в языке UML вводится в рассмотрение специальный элемент — концевая точка ассоциации или *конец ассоциации* (Association End), который графически соответствует точке соединения линии ассоциации с отдельным классом. Конец ассоциации является частью ассоциации, но не класса. Каждая ассоциация может иметь два или больше концов ассоциации. Наиболее важные свойства ассоциации указываются на диаграмме рядом с этими элементами ассоциации и должны перемещаться вместе с ними.

Одним из таких дополнительных обозначений является *имя роли* отдельного класса, входящего в ассоциацию. Имя роли представляет собой строку текста рядом с концом ассоциации для соответствующего класса. Она указывает специфическую роль, которую играет класс, являющийся концом рассматриваемой ассоциации. Имя роли не является обязательным элементом обозначений и может отсутствовать на диаграмме.

Следующий элемент обозначений — *кратность* ассоциации. Кратность относится к концам ассоциации и обозначается в виде интервала целых чисел, аналогично кратности атрибутов и операций классов, но без прямых скобок. Этот интервал записывается рядом с концом соответствующей ассоциации и означает потенциальное число отдельных экземпляров класса, которые могут иметь место, когда остальные экземпляры или объекты классов фиксированы.

Так, для рассмотренного ранее примера (рис. 5.3) кратность "1" для класса `Компания` означает, что каждый сотрудник может работать только в одной компании. Кратность "1..*" для класса `Сотрудник` означает, что в каждой компании могут работать несколько сотрудников, общее число которых заранее неизвестно и ничем не ограничено. Заметим, что вместо кратности "1..*" записать только символ "*" нельзя, поскольку последний означает кратность "0..*". Для данного примера это означало бы, что отдельные компании могут совсем не иметь сотрудников в своем штате. Но такая кратность вполне приемлема в других ситуациях, как это видно из другого рассмотренного выше примера (см. рис. 5.6), когда вполне возможно, что две выбранные футбольные команды не провели в сезоне ни одной встречи.

Что касается задания дополнительных свойств ассоциации, то в случае их наличия, они могут рассматриваться в качестве атрибутов класса ассоциации и быть указаны на диаграмме обычным для класса способом в соответствующей секции прямоугольника класса.

Ассоциация является наиболее общей формой отношения в языке UML. Все другие типы рассматриваемых отношений можно считать частным случаем данного отношения. Однако важность выделения специфических семантических свойств и дополнительных характеристик для других типов отношений обусловливают необходимость их самостоятельного изучения при построении диаграмм классов. Поскольку эти отношения имеют свои специальные обозначения и относятся к базовым понятиям языка UML, рассмотрим их последовательно.

5.2.2. Отношение обобщения

Отношение обобщения является обычным таксономическим отношением (или отношением классификации) между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком). Данное отношение может использоваться для представления иерархических взаимосвязей между пакетами, классами, вариантами использования и другими элементами языка UML.

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения. Согласно одному из главных принципов методологии ООАП — наследованию *(см. главу 1)*, класс-потомок обладает всеми свойствами и поведением класса-предка, а также имеет свои собственные свойства и поведение, которые могут отсутствовать у класса-предка. На диаграммах отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов (рис. 5.7). Стрелка указывает на более общий класс (класс-предок или суперкласс), а ее противоположный конец — на более специальный класс (класс-потомок или подкласс).

Рис. 5.7. Графическое изображение отношения обобщения в языке UML

От одного класса-предка одновременно могут наследовать несколько классов-потомков, что отражает таксономический характер данного отношения. В этом случае на диаграмме классов для подобного отношения обобщения указывается несколько линий со стрелками.

Например, класс *Геометрическая фигура* (курсив обозначает абстрактный класс) может выступать в качестве суперкласса для подклассов, соответствующих конкретным геометрическим фигурам, таким как Прямоугольник, Окружность, Эллипс и другим. Данный факт может быть представлен графически в форме диаграммы классов следующего вида (рис. 5.8).

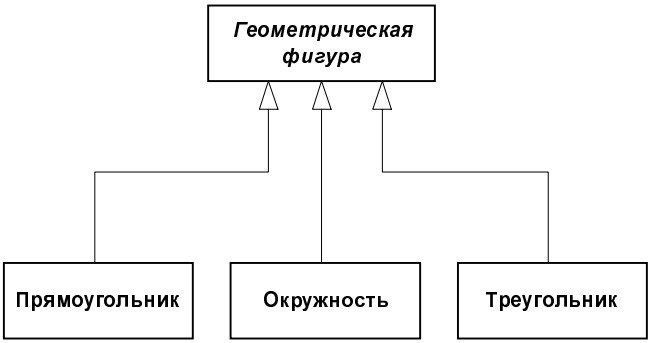


Рис. 5.8. Пример графического изображения отношения обобщения для нескольких классов-потомков

С целью упрощения обозначений на диаграмме классов и уменьшения числа стрелок-треугольников и совокупности линий, обозначающих одно и то же отношение обобщения, может быть просто изображена единственная стрелка. В этом случае отдельные линии изображаются сходящимися к единственной стрелке, которая имеет с этими линиями единственную точку пересечения (рис. 5.9).

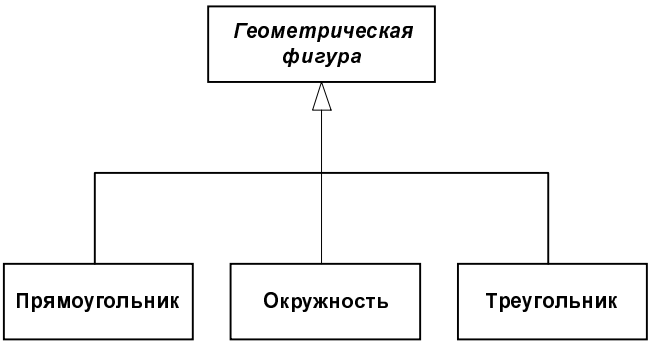


Рис. 5.9. Альтернативный вариант графического изображения отношения обобщения классов (рис. 5.8) для случая объединения отдельных линий

Графическое изображение отношения обозначения по форме соответствует графу специального вида, а именно — иерархическому дереву *(см. главу 2)*. Как нетрудно заметить, в этом случае класс-предок является корнем дерева, а классы-потомки — его листьями. Отличие заключается в возможности указания на диаграмме классов дополнительной семантической информации, которая может отражать различные теоретико-множественные характеристики данного отношения. При этом класс-предок на диаграмме может занимать произвольное положение относительно своих классов-потомков, определяемое лишь соображениями удобства.

В дополнение к простой стрелке-обобщения может быть присоединена строка текста, указывающая на некоторые специальные свойства этого отношения. Этот текст будет относиться ко всем линиям обобщения, которые идут к классам-потомкам. Поскольку отмеченное свойство касается всех подклассов данного отношения, спецификация этого свойства осуществляется в форме ограничения, которое должно быть записано в фигурных скобках.

В качестве ограничений могут быть использованы следующие ключевые слова языка UML.

- `{complete}` — означает, что в данном отношении обобщения специфицированы все классы-потомки, и других классов-потомков у данного класса-предка быть не может.
- `{incomplete}` — означает случай, противоположный первому. А именно, предполагается, что на диаграмме указанны не все классы-потомки. В последующем, возможно, разработчик восполнит их перечень, не изменяя уже построенную диаграмму.
- `{disjoint}` — означает, что классы-потомки не могут содержать объектов, одновременно являющихся экземплярами двух или более классов.
- `{overlapping}` — случай, противоположный предыдущему. А именно предполагается, что отдельные экземпляры классов-потомков могут принадлежать одновременно нескольким классам.

С учетом дополнительного использования стандартного ограничения диаграмма классов (рис. 5.9) может быть уточнена и специфицирована (рис. 5.10).

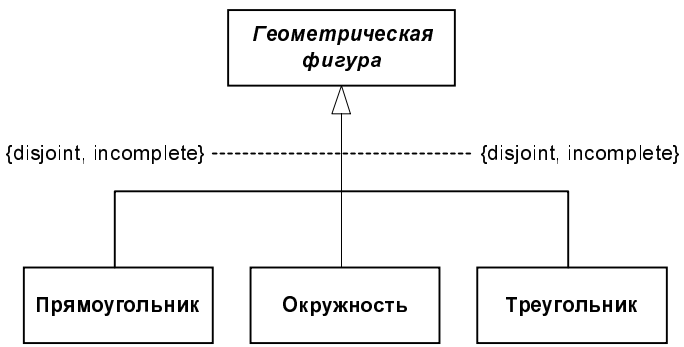


Рис. 5.10. Вариант уточненного графического изображения отношения обобщения классов с использованием строки-ограничения

Чтобы проиллюстрировать особенности использования отношения обобщения, преобразуем один из рассмотренных ранее примеров изображения классов в графическую нотацию языка UML. В качестве такого примера рассмотрим иерархию вложенности классов для абстрактного класса *Автомобиль* (см. рис. 1.2, 2.7). Как нетрудно заметить, отношение между отдельными

классами на этих рисунках есть именно отношение обобщения, которое в языке UML имеет специальное графическое обозначение. С учетом этой графической нотации, фрагмент семантической сети для представления иерархии класса-потомка *Автомобиль* (см. рис. 2.7) может быть представлен в виде следующей диаграммы классов (рис. 5.11).

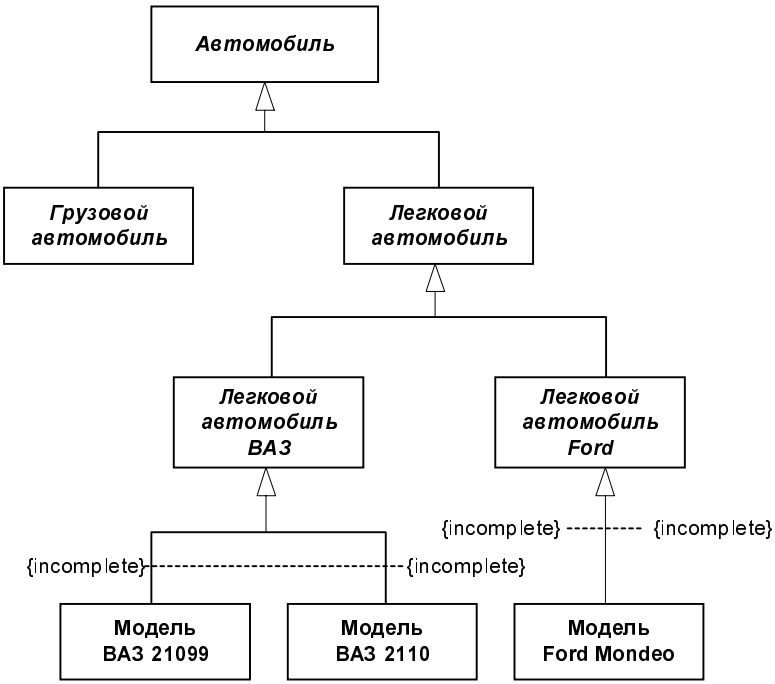


Рис. 5.11. Фрагмент диаграммы классов с отношением обобщения для представления иерархии классов *Автомобиль* из рассмотренного ранее примера (см. рис. 2.7)

Заметим, что в данном примере все классы верхних уровней являются абстрактными, т. е. не могут быть непосредственно использованы для создания экземпляров. Именно поэтому их имена записаны курсивом. В отличие от них классы нижнего уровня являются конкретными, поскольку могут быть представлены своими экземплярами, в качестве которых выступают изготовленные автомобили соответствующей модели с уникальным заводским номером.

Примечание

В качестве упражнения для закрепления рассмотренного материала можно предложить читателям построить диаграммы классов или хотя бы их фрагменты для библиотек стандартных классов MFC (Microsoft) и VCL (Borland)

с использованием графической нотации языка UML. Можно предположить, что в недалеком будущем справочные руководства по соответствующим средам программирования будут содержать изображения диаграмм классов в нотации языка UML, а возможно, и некоторые другие типы канонических диаграмм.

5.2.3. Отношение агрегации

Отношение агрегации имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, которая включает в себя в качестве составных частей другие сущности.

Данное отношение имеет фундаментальное значение для описания структуры сложных систем, поскольку применяется для представления системных взаимосвязей типа "часть-целое". Раскрывая внутреннюю структуру системы отношение агрегации показывает, из каких элементов состоит система и как они связаны между собой. С точки зрения модели отдельные части системы могут выступать как в виде элементов, так и в виде подсистем, которые, в свою очередь, тоже могут состоять из подсистем или элементов. Таким образом, данное отношение по своей сути описывает декомпозицию или разбиение сложной системы на более простые составные части, которые также могут быть подвергнуты декомпозиции, если в этом возникнет необходимость в последующем.

Примечание

В связи с рассмотрением данного отношения вполне уместно вспомнить о специальном термине "агрегат", который служит для обозначения технической системы, состоящей из взаимодействующих составных частей или подсистем. Эта аналогия не случайна и может служить для более наглядного понимания сути рассматриваемого отношения.

Очевидно, что рассматриваемое в таком аспекте деление системы на составные части также представляет собой некоторую иерархию, однако данная иерархия принципиально отличается от той, которая порождается отношением обобщения. Отличие заключается в том, что части системы никак не обязаны наследовать ее свойства и поведение, поскольку являются вполне самостоятельными сущностями. Более того, части целого обладают своими собственными атрибутами и операциями, которые существенно отличаются от атрибутов и операций целого.

Графически отношение агрегации изображается сплошной линией, один из концов которой представляет собой не закрашенный внутри ромб. Этот ромб указывает на тот из классов, который представляет собой "целое" или класс-контейнер. Остальные классы являются его "частями" (рис. 5.12).

В качестве примера отношения агрегации рассмотрим взаимосвязь типа "часть-целое", которая имеет место между классом *Грузовой автомобиль*

и такими его частями, как классы *Двигатель*, *Шасси*, *Кабина*, *Кузов*. Не претендуя на точное соответствие терминологии данной предметной области, нетрудно представить себе, что грузовой автомобиль состоит из двигателя, шасси, кабины и кузова. Именно это отношение и описывает отношение агрегации.

Рис. 5.12. Графическое изображение отношения агрегации в языке UML

Еще одним примером отношения агрегации может служить известное каждому из читателей разделение персонального компьютера на составные части: системный блок, монитор, клавиатуру и мышь. Используя обозначения языка UML, компонентный состав ПК можно представить в виде соответствующей диаграммы классов (рис. 5.13), которая в данном случае иллюстрирует отношение агрегации.

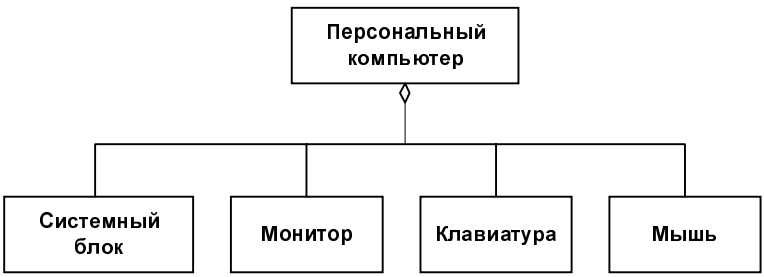


Рис. 5.13. Диаграмма классов для иллюстрации отношения агрегации на примере ПК

5.2.4. Отношение композиции

Отношение композиции является частным случаем отношения агрегации. Это отношение служит для спецификации более сильной формы отношения "часть-целое", при которой составляющие части тесно взаимосвязаны с целым. Специфика этой взаимосвязи заключается в том, что части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части.

Графически отношение композиции изображается сплошной линией, один из концов которой представляет собой закрашенный внутри ромб. Этот ромб указывает на тот из классов, который представляет собой класс-композит. Остальные классы являются его "частями" (рис. 5.14).

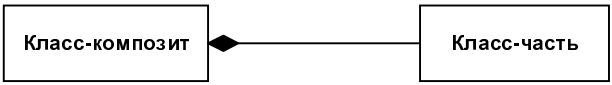


Рис. 5.14. Графическое изображение отношения композиции в языке UML

Возможно, самый наглядный пример этого отношения представляет собой живая клетка в биологии, в отрыве от которой не могут существовать ее составные части. Другой пример — окно графического интерфейса программы, которое может состоять из строки заголовка, кнопок управления размером, полос прокрутки, главного меню, рабочей области и строки состояния. Нетрудно заключить, что подобное окно представляет собой класс, а его элементы могут являться как классами, так и атрибутами или свойствами этого окна. Последнее обстоятельство весьма характерно для отношения композиции, поскольку отражает различные способы представления данного отношения.

Для отношений композиции и агрегации могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно, могут указываться кратности отдельных классов, которые в общем случае не являются обязательными. Применительно к описанному выше примеру класс `Окно программы` является классом-композитом, а взаимосвязи составляющих его частей могут быть изображены следующей диаграммой классов (рис. 5.15).

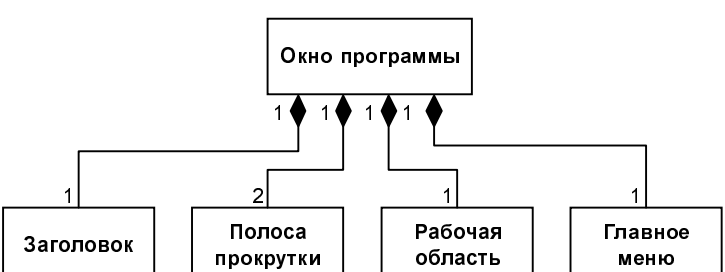


Рис. 5.15. Диаграмма классов для иллюстрации отношения композиции на примере класса-композита `Окно программы`

Примечание

Данный пример может иллюстрировать и другие особенности разрабатываемой компьютерной программы, которые не указывались в явном виде при описании этого примера. Так, в частности, указание кратности 1 рядом с классом `Рабочая область` может характеризовать одно документное приложение.

5.2.5. Отношение зависимости

Отношение зависимости было рассмотрено ранее *(см. главу 4)* и в общем случае указывает некоторое семантическое отношение между двумя элементами модели или двумя множествами таких элементов, которое не является отношением ассоциации, обобщения или реализации. Оно касается только самих элементов модели и не требует множества отдельных примеров для пояснения своего смысла. Отношение зависимости используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависящего от него элемента модели.

Отношение зависимости графически изображается пунктирной линией между соответствующими элементами со стрелкой на одном из ее концов. На диаграмме классов данное отношение связывает отдельные классы между собой, при этом стрелка направлена от класса-клиента зависимости или зависимого класса к классу-источнику или независимому классу (рис. 5.16). На данном рисунке изображены два класса: `Класс А` и `Класс Б`, при этом `Класс Б` является источником некоторой зависимости, а `Класс А` — клиентом этой зависимости.

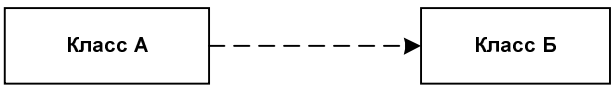


Рис. 5.16. Графическое изображение отношения зависимости на диаграмме классов

Наиболее часто в качестве классов-клиентов и классов-источников зависимости могут выступать целые множества классов, организованные в пакеты. В этом случае стрелка должна быть направлена от пакета-клиента зависимости или зависимого пакета к пакету-источнику или независимому пакету (рис. 5.17). На данном рисунке изображены два пакета: Пакет А и Пакет Б, при этом Пакет Б является источником некоторой зависимости, а Пакет А — клиентом этой зависимости. Наличие у зависимости дополнительного стереотипа <<access>> служит для обозначения доступности открытых атрибутов и операций классов Пакета Б для классов Пакета А, но не наоборот.

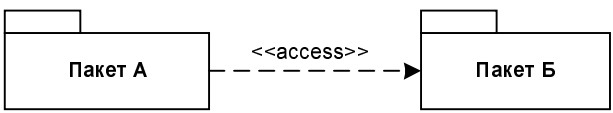


Рис. 5.17. Графическое представление зависимости между пакетами с дополнительным стереотипом <<access>>

Стрелка зависимости может помечаться и другими необязательными, но стандартными ключевыми словами в кавычках. Для отношения зависимости

в спецификации языка UML определены стереотипы для специальных случаев зависимостей. Эти стереотипы записываются рядом со стрелкой, которая соответствует данной зависимости. Стандартные стереотипы, которые могут использоваться с отношением зависимости, перечислены ниже.

- <<bind>> — класс-клиент может использовать некоторый шаблон для своей последующей параметризации.
- <<derive>> — атрибуты класса-клиента могут быть вычислены по атрибутам класса-источника.
- <<import>> — открытые атрибуты и операции класса-источника становятся частью класса-клиента, как если бы они были объявлены непосредственно в нем.
- <<refine>> — указывает, что класс-клиент служит уточнением класса-источника в силу причин исторического характера, например, при появлении некоторой дополнительной информации в ходе работы над проектом.

5.3. Расширение языка UML для построения моделей программного обеспечения и бизнес-систем

Как уже отмечалось ранее, одним из несомненных достоинств языка UML является наличие механизмов расширения, которые позволяют ввести в рассмотрение дополнительные графические обозначения, ориентированные для решение задач из определенной предметной области. При этом в самой документации на язык UML представлено два специальных примера подобных расширений: профиль для процесса разработки программного обеспечения (The UML Profile for Software Development Processes) и профиль для бизнес-моделирования (The UML Profile for Business Modeling).

В рамках первого из них предложено три специальных графических примитива, которые могут быть использованы для уточнения семантики отдельных классов при построении различных диаграмм.

- *Управляющий класс* (control class), отвечающий за координацию действий других классов. У каждого варианта использования и диаграммы классов должен быть хотя бы один управляющий класс, контролирующий последовательность выполнения действий этого варианта использования. Как правило, этот класс является активным и инициирует рассылку множества сообщений другим классам модели. Кроме специального обозначения управляющий класс может быть изображен в форме обычного прямоугольника класса со стереотипом <<control>> (рис. 5.18, *а*).

- *Класс-сущность* (entity) содержит информацию, которая должна храниться постоянно и не уничтожаться с уничтожением объектов данного класса или прекращением работы моделируемой системы (выключением системы или завершением программы). Этот класс может соответствовать отдельной таблице базы данных, в этом случае его атрибуты могут быть полями таблицы, а операции — присоединенными процедурами. Как правило, этот класс является пассивным и лишь принимает сообщения от других классов модели. Класс-сущность может быть изображен также стандартным образом в форме обычного прямоугольника класса со стереотипом <<entity>> (рис. 5.18, *б*).
- *Граничный класс* (boundary) располагается на границе системы с внешней средой, но является составной частью системы. Граничный класс может быть изображен также стандартным образом в форме обычного прямоугольника класса со стереотипом <<boundary>>(рис. 5.18, *в*).

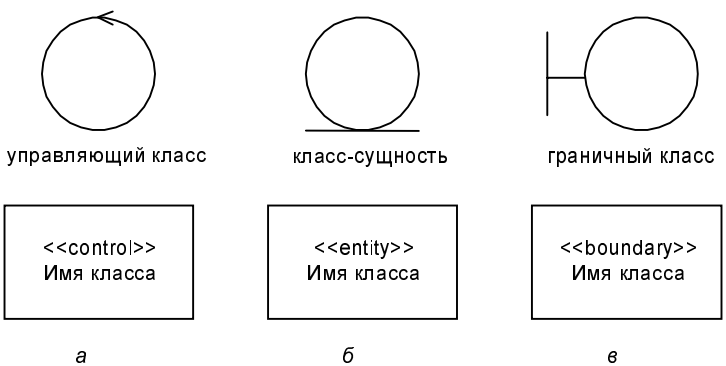


Рис. 5.18. Графическое изображение классов для моделирования программного обеспечения

В рамках второго профиля также предложено три специальных графических примитива, которые могут быть использованы для уточнения семантики отдельных классов при построении моделей бизнес-систем.

- *Сотрудник* (worker) — класс, который служит для представления в бизнес-системе любого сотрудника, который является элементом бизнес-системы и взаимодействует с другими сотрудниками при реализации бизнес-процесса. Этот класс также может быть изображен в форме обычного прямоугольника класса со стереотипом <<worker>> или <<internalWorker>> (рис. 5.19, *а*).
- *Сотрудник для связи с окружением* (caseworker) — класс, который служит для представления в бизнес-системе такого сотрудника, который, являясь элементом бизнес-системы, непосредственно взаимодействует с актерами (бизнес-актерами) при реализации бизнес-процесса. Этот класс также

может быть изображен в форме обычного прямоугольника класса со стереотипом <<caseWorker>> (рис. 5.19, *б*).

- *Бизнес-сущность* (business entity) является специальным случаем класса-сущности, который также не инициирует никаких сообщений. Этот класс служит для сохранения информации о результатах выполнения бизнес-процесса в моделируемой бизнес-системе (организации). Этот класс также может быть изображен в форме обычного прямоугольника класса со стереотипом <<business entity>> (рис. 5.19, *в*).

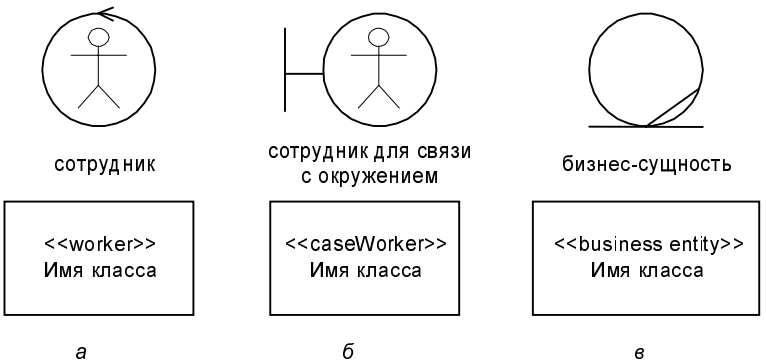


Рис. 5.19. Графическое изображение классов для моделирования бизнес-систем

Примечание

Строго говоря, графическое изображение бизнес-сущности (рис. 5.19, *в*) несколько отличается от того, которое представлено в документации по языку UML. Указанное обозначение присутствует в инструменте IBM Rational Rose и по внутренней логике изложения должно было бы войти в спецификацию языка UML. Поскольку этого сделано не было, остается только догадываться о том, что это — опечатка разработчиков или их нежелание следовать рекомендациям одного из CASE-средств?

5.3.1. Интерфейс

Интерфейс (interface) в контексте языка UML является специальным случаем класса, у которого имеются только операции и отсутствуют атрибуты. Для обозначения интерфейса используется специальный графический символ окружность или стандартный способ — прямоугольник класса со стереотипом <<interface>> (рис. 5.20).

На диаграмме вариантов использования интерфейс изображается в виде маленького круга, рядом с которым записывается его имя (рис. 5.20, *а*). В качестве имени может быть существительное, которое характеризует соответ-

ствующую информацию или сервис (например, "Датчик температуры", "Форма ввода", "Сирена", "Видеокамера") (рис. 5.20, *б*). С учетом языка реализации модели имя интерфейса, как и имена других классов, рекомендуется записывать на английском. В этом случае оно должно начинаться с заглавной буквы I, например, ItemperatureSensor, IsecureInformation (рис. 5.20, *в*).

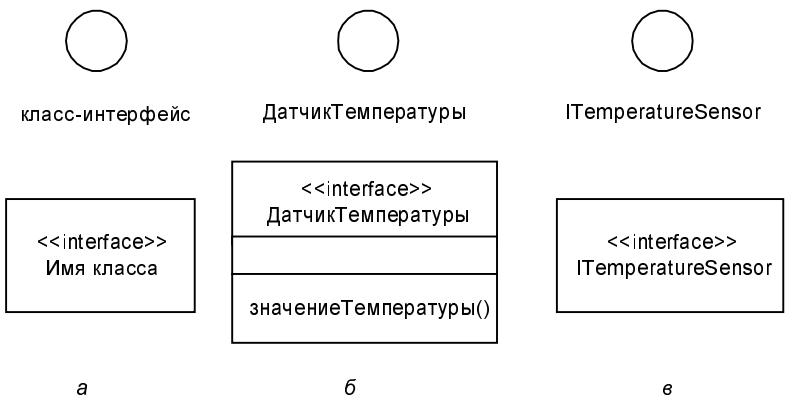


Рис. 5.20. Примеры графического изображения интерфейсов на диаграмме классов

Интерфейсы на диаграмме служат для спецификации таких элементов модели, которые видимы извне, но их внутренняя структура остается скрытой от клиентов. В языке UML интерфейс является классификатором и характеризует только ограниченную часть поведения моделируемой сущности. Применительно к диаграммам классов, интерфейсы определяют совокупность операций, которые обеспечивают необходимый набор сервисов или функциональности для актеров. Интерфейсы не могут содержать ни атрибутов, ни состояний, ни направленных ассоциаций. Они содержат только операции без указания особенностей их реализации. Формально интерфейс эквивалентен абстрактному классу без атрибутов и методов с наличием только абстрактных операций.

Примечание

Имена интерфейсов подчиняются общим правилам наименования классов языка UML, т. е. имя может состоять из любого числа букв, цифр и некоторых знаков препинания, таких как двойное двоеточие "::". Последний символ используется для более сложных имен, включающих в себя не только имя самого интерфейса (после знака), но и имя пакета, который включает в себя данный интерфейс (перед знаком). Примерами таких имен являются: "`СетьПредприятия :: Брандмауэр`" — для указания класса, представляющего специальный защитный сервер сети предприятия или "`СистемаАутентификацииКлиентов :: ФормаВводаПароля`".

С системно-аналитической точки зрения интерфейс не только отделяет спецификацию операций системы от их реализации, но и определяет общие границы проектируемой системы. В последующем интерфейс может быть уточнен явным указанием тех операций, которые специфицируют отдельный аспект поведения системы. Графическое изображение интерфейсов в форме окружности может использоваться и на других типах канонических диаграмм, например, диаграммах развертывания и диаграммах вариантов использования.

Важность интерфейсов заключается в том, что они определяют стыковочные узлы в проектируемой системе, что совершенно необходимо для организации коллективной работы над проектом. Более того, спецификация интерфейсов способствует "безболезненной" модификации уже существующей системы при переходе на новые технологические решения. В этом случае изменению подвергается только реализация операций, но никак не функциональность самой системы. А это обеспечивает совместимость последующих версий программ с первоначальными при спиральной технологии разработки программных систем.

5.4. Шаблоны или параметризованные классы

Шаблон (template) или *параметризованный класс* (parametrized class) предназначен для обозначения такого класса, который в своем описании имеет несколько формальных параметров. Он определяет целое семейство или множество классов, каждый из которых может быть получен связыванием этих параметров с действительными значениями. Обычно параметрами шаблонов служат типы атрибутов классов, такие как целые числа, перечисление, массив строк и другие. В более сложном случае формальные параметры могут представлять и операции класса.

Графически шаблон изображается прямоугольником класса, на верхний правый угол которого наложен маленький прямоугольник из пунктирных линий (рис. 5.20, *а*). Основной прямоугольник может быть разделен на секции, аналогично обозначению для класса. В верхнем прямоугольнике указывается список формальных параметров для тех классов, которые могут быть получены на основе данного шаблона. В верхней секции самого шаблона записывается его имя по правилам записи имен для классов.

Шаблон не может быть непосредственно использован в качестве класса, поскольку содержит неопределенные параметры. Чаще всего в качестве шаблона выступает некоторый суперкласс, параметры которого уточняются в производных классах. В этом случае между ними существует некоторая взаимосвязь, которая в языке UML называется отношением *реализации*. Это отношение изображается пунктирной линией со стрелкой в форме

треугольника, возможно с дополнительным стереотипом <<bind>> (рис. 5.21, *б*). Эта взаимосвязь означает, что класс-клиент может использовать некоторый шаблон для своей последующей параметризации. Приведенный пример отражает тот факт, что класс `Адрес` может быть получен из шаблона `СвязныйСписок` на основе замены формальных параметров "S, k, l" фактическими параметрами "`улица, дом, квартира`".

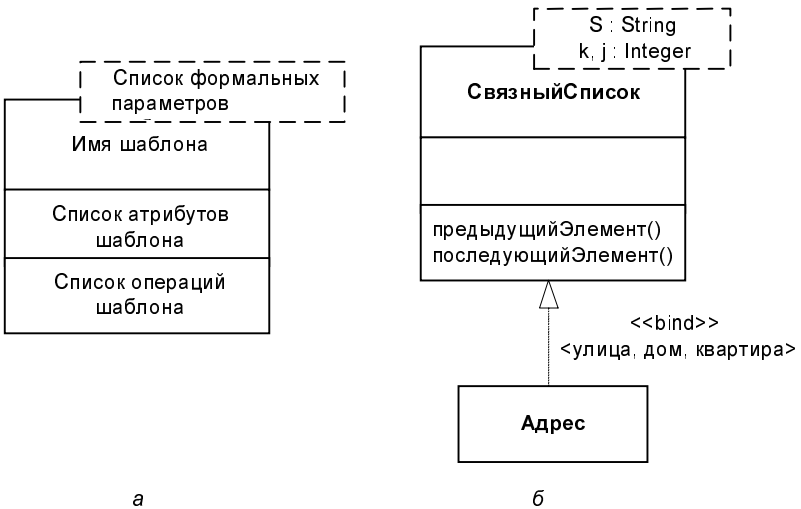


Рис. 5.21. Графическое изображение шаблона на диаграмме классов

С другой стороны, этот же шаблон может использоваться для задания (инстанцирования) другого класса, скажем, класса `ТочкиНаПлоскости`. В этом случае класс `ТочкиНаПлоскости` актуализирует те же формальные параметры, но с другими значениями, например, `<координатыТочки, x, y>`.

Примечание

В частном случае между шаблоном и его классом-клиентом может иметь место отношение обобщения с наследованием свойств шаблона.

Концепция шаблонов является достаточно мощным средством в ООП, и поэтому ее использование в языке UML позволяет не только сократить размеры диаграмм моделей программных систем, но и наиболее корректно управлять наследованием свойств и поведения отдельных элементов модели.

5.5. Пример построения диаграммы классов системы управления банкоматом

Продолжая построение модели сквозного примера модели системы управления банкоматом, построим соответствующую диаграмму классов. Вообще говоря, подобная диаграмма классов может быть разработана различным образом, что отражает субъективный характер собственно разработчика на отдельные аспекты реализации модели. Один из возможных вариантов диаграммы классов для данного примера изображен на рис. 5.22.

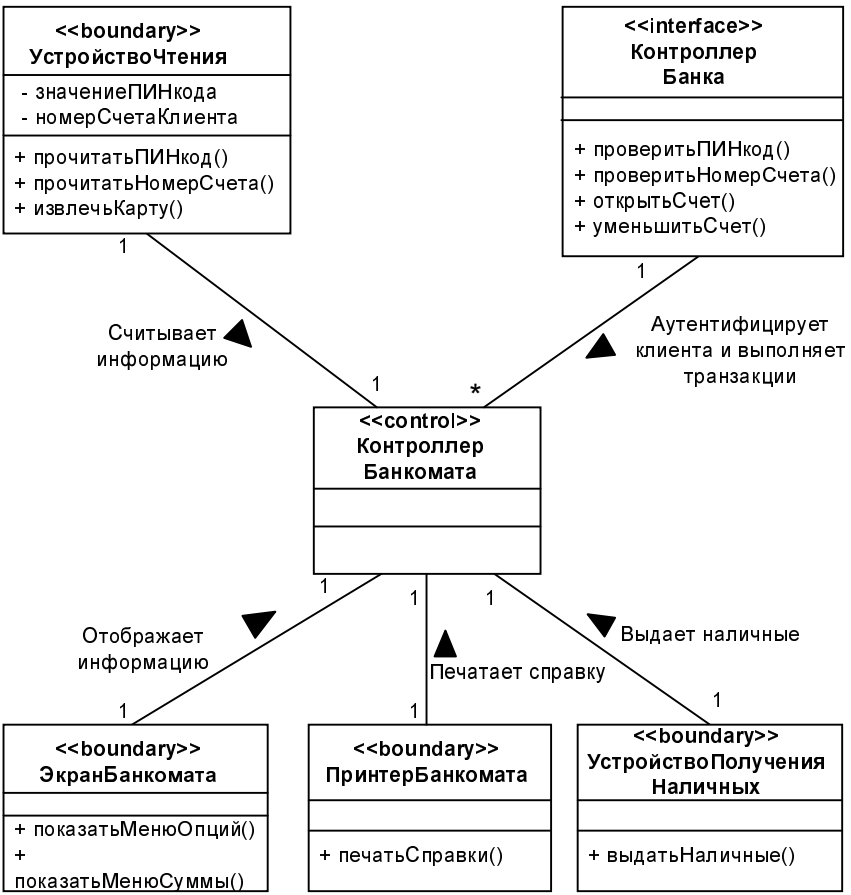


Рис. 5.22. Диаграмма классов системы управления банкоматом

Данная диаграмма содержит 6 классов, 4 из которых представлены в форме граничных классов, 1 — как управляющий класс и 1 — как класс-

интерфейс. Поскольку класс `КонтроллерБанка` не является предметом разработки, а только лишь предоставляет часть своих операций в качестве сервиса, он может быть представлен в форме интерфейса, т. е. со стереотипом <<interface>>. Что касается состава операций классов и их видимостей, то их выбор определяется исходя из возможной программной реализации данной модели.

Изображенная диаграмма классов является предварительной статической моделью системы управления банкоматом и может уточняться на последующих этапах работы над проектом. В частности, может потребоваться изменить состав операций отдельных классов или их видимость, ввести дополнительные атрибуты или более детально расписать сигнатуру операций. Все эти действия предлагается выполнить читателям самостоятельно в качестве упражнения.

Примечание

Если предполагается программная реализация модели проектируемой системы, то, очевидно, имена классов в этом случае следует записывать на английском, при этом возможно учитывать и синтаксические особенности языка реализации. В рассмотренном примере имена классов записаны на русском из методических соображений.

5.6. Рекомендации по построению диаграмм классов

Процесс разработки диаграммы классов занимает центральное место при разработке проектов сложных систем. От умения правильно выбрать классы и установить между ними взаимосвязи часто зависит не только успех процесса проектирования, но и производительность выполнения программы. Как показывает практика ООП, каждый программист в своей работе стремится в той или иной степени использовать уже накопленный личный опыт при разработке новых проектов. Это обусловлено желанием свести новую задачу к уже решенным, чтобы иметь возможность использовать не только проверенные фрагменты программного кода, но и отдельные компоненты в целом (библиотеки компонентов).

Такой стереотипный подход позволяет существенно сократить сроки реализации проекта, однако приемлем лишь в том случае, когда новый проект концептуально и технологически не слишком отличается от предыдущих. В противном случае платой за сокращение сроков проекта может стать его реализация на устаревшей технологической базе. Что касается собственно объектной структуризации при построении модели предметной области, то здесь уместно придерживаться тех рекомендаций, которые накоплены в ООП. Они широко освещены в литературе и поэтому здесь не рассматриваются.

При определении классов, атрибутов и операций и задании их имен и типов перед отечественными разработчиками всегда встает невольный вопрос: какой из языков использовать в качестве естественного — русский или английский? С одной стороны, использование родного языка для описания модели является наиболее естественным способом ее представления и в наибольшей степени отражает коммуникативную функцию модели системы. С другой стороны, разработка модели является лишь одним из этапов разработки соответствующей системы, а применение инструментальных средств для ее реализации в абсолютном большинстве случаев требует использования англоязычных терминов. Именно поэтому возникает характерная неоднозначность, с которой, по-видимому, совершенно незнакома англоязычная аудитория.

Отвечая на поставленный выше вопрос, следует отметить, что наиболее целесообразно придерживаться следующих рекомендаций. При построении диаграммы вариантов использования, являющейся наиболее общей концептуальной моделью проектируемой системы, применение русскоязычных терминов является не только оправданным с точки зрения описания структуры предметной области, но и эффективным с точки зрения коммуникативного взаимодействия с заказчиком и пользователями. При построении остальных типов диаграмм следует придерживаться разумного компромисса.

В частности, на начальных этапах разработки диаграмм целесообразно использовать русскоязычные термины, что вполне очевидно и оправданно. Однако, по мере готовности графической модели для реализации в виде программной системы и передачи ее для дальнейшей работы программистам, акцент может смещаться в сторону использования англоязычных терминов, которые в той или иной степени отражают особенности языка программирования, на котором предполагается реализация данной модели.

Использование CASE-средств для автоматизации ООАП, чаще всего, накладывает свои собственные требования на язык спецификации моделей. В частности, отдельные инструменты могут не иметь корректного отображения символов кириллицы. Именно по этой причине большинство примеров в литературе даются в англоязычном представлении, а при их переводе на русский может быть утрачена не только точность формулировок, но и семантика соответствующих понятий.

При разработке диаграмм классов существенную помощь могут оказать получившие широкое освещение в литературе паттерны и антипаттерны анализа и проектирования, которые обобщают положительный (соответственно, отрицательный) опыт специалистов на основе уже выполненных проектов.

После разработки диаграммы классов процесс ООАП может быть продолжен в двух направлениях. С одной стороны, если поведение системы тривиально, то можно приступить к разработке диаграмм кооперации и последовательности. К изучению особенностей построения диаграмм кооперации

мы и приступим в следующей главе. Однако для сложных динамических систем, в частности для параллельных систем и систем реального времени, важнейшим аспектом их функционирования является поведение. Особенности поведения таких систем могут быть представлены на диаграммах состояний и деятельности, разработка которых в общем случае не является обязательной, а определяется спецификой разрабатываемого проекта.

В заключение приводится сводка всех рассмотренных графических обозначений, которые могут встретиться при построении диаграмм вариантов использования (табл. 5.1). Хотя в языке UML имеются некоторые другие элементы, данного перечня может оказаться вполне достаточно для выполнения большинства реальных проектов.

Таблица 5.1. *Графические элементы диаграмм классов*

Графическое изображение	Название
Имя класса Атрибуты класса Операции класса	Класс (class)
	Управляющий класс (control class)
	Класс-сущность (entity class)
	Граничный класс (boundary class)
	Сотрудник (worker)
	Сотрудник для связи с окружением (caseworker)

Таблица 5.1 *(окончание)*

Графическое изображение	Название
	Бизнес-сущность (business entity)
	Пакет (package)
	Класс-шаблон (template)
	Интерфейс (interface)
	Ассоциация (association relationship)
	Обобщение (generalization relationship)
	Агрегирование (aggregation relationship)
	Композиция (composition relationship)
	Зависимость (dependency relationship)

Глава 6

Диаграмма кооперации (collaboration diagram)

Диаграмма классов представляет собой логическую модель статического представления моделируемой системы. Действительно, на данной диаграмме могут быть изображены только взаимосвязи структурного характера между классами, не зависящие от времени или реакции системы на внешние события. Однако различные составные элементы систем не существуют изолированно, а оказывают определенное влияние друг от друга, что и отличает систему как целостное образование от простой совокупности элементов.

В языке UML взаимодействие элементов рассматривается в информационном аспекте, т. е. объекты обмениваются между собой некоторой информацией. При этом информация передается в форме так называемых сообщений, играющих роль своеобразных стимулов. Другими словами, хотя сообщение и имеет информационное содержание, оно приобретает дополнительное свойство оказывать направленное влияние на своего получателя. Это полностью согласуется с принципами ООАП, когда любые виды информационного взаимодействия между элементами модели должны быть сведены к отправке и приему сообщений между ними.

Для моделирования взаимодействия объектов в языке UML используются соответствующие диаграммы *взаимодействия*. Говоря об этих диаграммах, имеют в виду два различных способа визуализации взаимодействия.

Во-первых, можно рассматривать взаимодействие объектов в контексте статической структуры модели. В этом случае для представления структурных особенностей передачи и приема сообщений между объектами используется диаграмма кооперации. Этот вид канонических диаграмм является предметом рассмотрения настоящей главы.

Во-вторых, взаимодействие объектов можно рассматривать во времени, и тогда для представления временных особенностей передачи и приема сообщений между объектами используется диаграмма последовательности. Этот вид канонических диаграмм рассматривается в *главе 7*.

Рассматривая диаграммы взаимодействия, следует отметить, что диаграмма кооперации предназначена для спецификации структурных аспектов

взаимодействия, а диаграмма последовательности служит для визуализации временных аспектов взаимодействия. В то же время характерной особенностью диаграммы кооперации является возможность графического представления не только необходимых структурных отношений между объектами, участвующими в этом взаимодействии, но также визуализация последовательности сообщений, описывающих собственно процесс динамического взаимодействия объектов.

Таким образом, на диаграмме кооперации поведение системы описывается на уровне отдельных объектов, которые обмениваются между собой сообщениями, чтобы достичь нужной цели или реализовать некоторый вариант использования. С точки зрения аналитика или архитектора системы важно представить в проекте структурные связи отдельных объектов между собой. Такое представление структуры модели как совокупности взаимодействующих объектов и обеспечивает диаграмма кооперации.

6.1. Кооперация

Понятие *кооперации* (collaboration) является одним из фундаментальных понятий в языке UML. Оно служит для обозначения множества взаимодействующих с определенной целью объектов в общем контексте моделируемой системы. Цель самой кооперации состоит в том, чтобы специфицировать особенности реализации отдельных вариантов использования или наиболее значимых операций в системе. Кооперация определяет структуру поведения системы в терминах взаимодействия участников этой кооперации.

Кооперация может быть представлена на двух уровнях:

- на уровне спецификации — показывает роли классификаторов и роли ассоциаций в рассматриваемом взаимодействии;
- на уровне примеров (экземпляров) — указывает объекты и связи, образующие отдельные роли в кооперации.

Далее рассматриваются особенности изображения диаграмм кооперации для этих двух уровней представления.

6.1.1. Диаграмма кооперации уровня спецификации

Диаграмма кооперации уровня спецификации показывает роли, которые играют участвующие во взаимодействии элементы. Элементами кооперации на этом уровне являются множества объектов (классы) и множества связей (ассоциации), которые обозначают отдельные роли классификаторов и роли ассоциаций между участниками кооперации.

Кооперация на уровне спецификации изображается на диаграмме пунктирным эллипсом, внутри которого записывается имя этой кооперации

(рис. 6.1). Такое представление кооперации может относиться к отдельному варианту использования и детализировать особенности его последующей реализации. Символ эллипса кооперации соединяется отрезками пунктирных линий с каждым из участников этой кооперации, в качестве которых могут выступать множество объектов или классы. Каждая из этих пунктирных линий помечается ролью (role) участника. Роли соответствуют именам элементов в контексте всей кооперации. Эти имена трактуются как параметры, которые ограничивают спецификацию элементов при любом их появлении в отдельных представлениях модели.

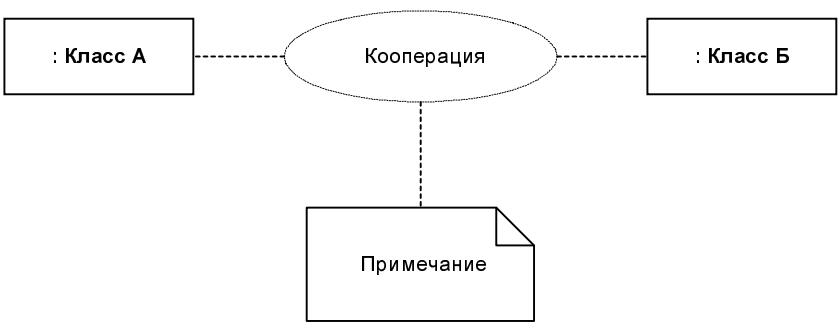


Рис. 6.1. Общее представление кооперации на диаграммах уровня спецификации

Множество объектов на диаграмме кооперации уровня спецификации обозначается прямоугольником класса, внутри которого записывается строка текста в следующем формате:

```
  '/'<Имя роли классификатора>':
'<Имя классификатора>[','<Имя классификатора>]* .
```

`Имя роли классификатора` указывает на роль в рассматриваемой кооперации, которую играет соответствующее множество объектов, созданных на базе класса `Имя классификатора`. Если роль, которую должны играть объекты, наследуется от нескольких классов, то все они должны быть указаны явно и разделены запятой. `Имя классификатора`, если это необходимо, может включать полный путь всех вложенных пакетов. При этом один пакет от другого отделяется двойным двоеточием "::". Если не возникает путаницы, можно ограничиться указанием только ближайшего из пакетов, которому принадлежит данный классификатор. Символ "*" применяется для указания возможности итеративного повторения имени классификатора.

Имя роли может быть опущено в том случае, если существует только одна роль в кооперациях, которую могут играть объекты, созданные на базе класса `Имя классификатора`, либо если оно с очевидностью следует из контекста диаграммы. В этом случае оно исключается из строки текста вместе с предшествующим ему символом "/".

Ниже приводятся возможные варианты записи имен множества объектов на диаграммах кооперации уровня спецификации:

- / R : C — роль R на основе классификатора C;
- / R — роль R без указания имени классификатора;
- : C — анонимная роль на базе классификатора C.

Отдельные примеры изображения объектов с ролями объектов и именами классов на диаграммах кооперации уровня спецификации приводятся на рис. 6.2.

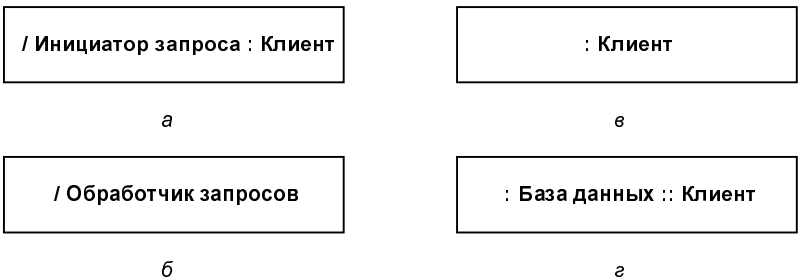


Рис. 6.2. Примеры графического изображения объектов на диаграммах кооперации уровня спецификации

Здесь в первом случае (рис. 6.2, *а*) обозначено множество объектов, играющих роль `Инициатор запроса`, образуемых от класса с именем `Клиент`. Далее (рис. 6.2, *б*) следует обозначение роли `Обработчик запросов`. В следующих вариантах записи (рис. 6.2, *в*, *г*) присутствует только имя класса `Клиент`, при этом соответствующие роли не указаны. Последний случай характерен для ситуации, когда в модели могут присутствовать несколько классов с именем `Клиент`, поэтому требуется явно указать имя соответствующего пакета `База данных` (рис. 6.2, *г*). При этом имя пакета отделяется от имени класса двойным двоеточием.

Если кооперация допускает обобщенное представление, то на диаграммах могут быть указаны отношения обобщения соответствующих элементов. Этот способ может быть использован для определения отдельных коопераций, которые являются, в свою очередь, частным случаем или специализацией другой кооперации. Такая ситуация изображается обычной стрелкой обобщения, направленной от символа дочерней кооперации к символу кооперации-предка. При этом роли дочерних коопераций могут быть специализациями ролей коопераций-предков, как изображено на следующем рисунке (рис. 6.3).

В отдельных случаях возникает необходимость явно указать тот факт, что кооперация является реализацией некоторой операции классификатора. Это можно представить одним из двух способов.

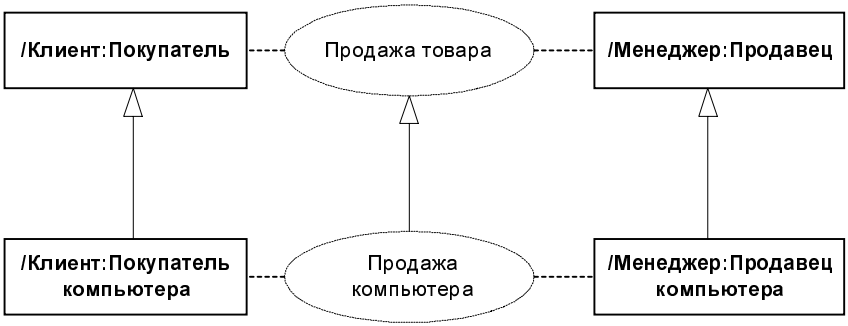


Рис. 6.3. Пример графического изображения отношения обобщения между отдельными кооперациями уровня спецификации

Во-первых, можно соединить символ кооперации пунктирной линией со стрелкой с символом класса, реализацию операции которого специфицирует данная кооперация. Так, если в качестве класса рассмотреть `ЗаказНаПокупкуТовара`, у которого имеется операция `оформитьЗаказ()`, то ее реализация может быть специфицирована в форме кооперации (рис. 6.4, *а*).

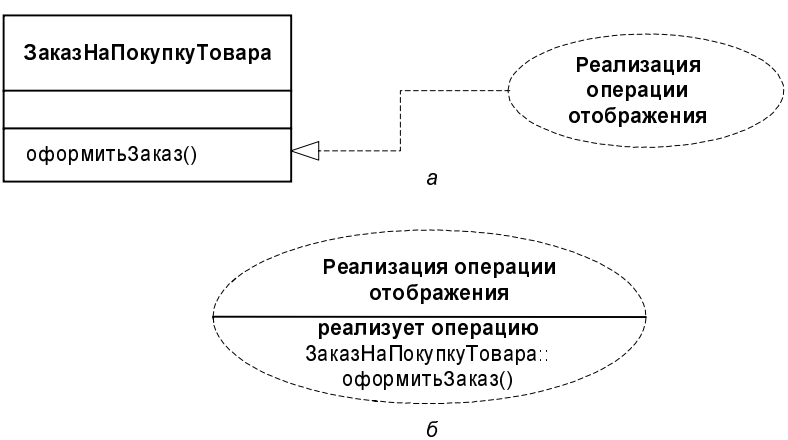


Рис. 6.4. Альтернативные способы представления реализации операции класса

Во-вторых, можно просто изобразить символ кооперации, внутри которого указать всю необходимую информацию, записанную по определенным правилам (рис. 6.4, *б*).

Представление кооперации на уровне спецификации используется на начальных этапах проектирования. В последующем каждая из кооперций подлежит детализации на уровне примеров (экземпляров), на котором раскрывается содержание и структура взаимосвязей ее элементов на отдельной диаграмме кооперации. При этом в качестве элементов диаграммы кооперации

выступают объекты и связи, дополненные сообщениями. Именно эти элементы модели являются предметом дальнейшего рассмотрения в настоящей главе.

6.1.2. Диаграмма кооперации уровня примеров

Диаграмма кооперации уровня примеров (instances) визуализирует объекты (экземпляры классов), связи (экземпляры ассоциаций) и сообщения. При этом связи дополняются стрелками сообщений. На этом уровне показываются только те объекты, которые имеют непосредственное отношение к реализации моделируемой кооперации. Поскольку эта разновидность нашла наибольшее применение при выполнении реальных проектов, в дальнейшем внимание сосредоточено на рассмотрении особенностей ее графического представления.

На диаграмме кооперации уровня примеров прежде всего изображаются участвующие во взаимодействии объекты. При этом вовсе не обязательно изображать объекты всех классов модели. Далее, как и на диаграмме классов, могут быть показаны структурные отношения между объектами в виде различных соединительных линий. При этом для связей можно явно указать имена ролей, которые играют объекты в данной взаимосвязи.

Важно помнить, что в то время как классификатор требует полного описания всех своих экземпляров, роль классификатора или объект требует описания только тех связей, которые необходимы для реализации отдельной кооперации. И, наконец, изображаются динамические взаимосвязи — потоки сообщений. Они представляются в форме стрелок с указанием направления рядом с соединительными линиями между объектами, при этом указываются имена сообщений и их порядковые номера в общей последовательности сообщений.

Отсюда вытекает важное следствие. Одна и та же совокупность объектов может участвовать в реализации различных коопераций. При этом, в зависимости от рассматриваемой кооперации, могут изменяться как связи между отдельными объектами, так и поток сообщений между ними. Именно это отличает диаграмму кооперации от диаграммы классов, на которой должны быть указаны все без исключения классы, их атрибуты и операции, а также все ассоциации и другие структурные отношения между элементами модели.

Ниже представлен начальный фрагмент диаграммы кооперации уровня примеров для модели телефонного разговора с использованием обычной телефонной сети (рис. 6.5). Объектами в этом примере являются: два абонента *a* и *b*, два телефонных аппарата *c* и *d*, коммутатор и сам разговор как объект моделирования. В данном случае абонентов целесообразно рассматривать как актеров, причем первый из них *a* — играет активную роль, а второй актер *b* — пассивную роль. Заметим также, что коммутатор и разговор являются анонимными объектами, причем телефонный аппарат *c* и комму-

татор изображены как активные объекты, а телефонный аппарат d и разговор — как пассивные. Особенности изображения подобных объектов будут рассмотрены далее.

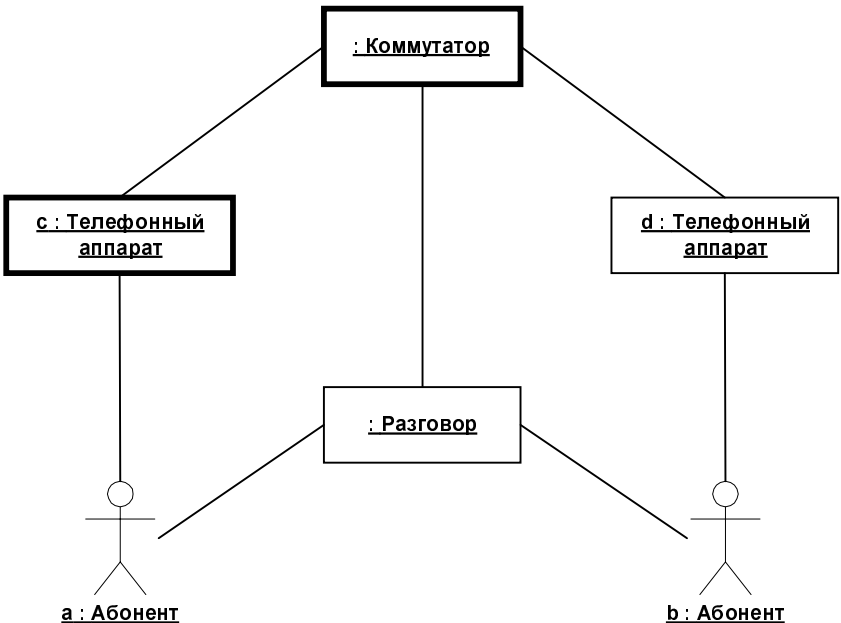


Рис. 6.5. Начальный фрагмент диаграммы кооперации уровня примеров для модели обычного телефонного разговора

В последующем этот фрагмент диаграммы уровня примеров будет дополнен сообщениями между объектами. Соответствующая диаграмма кооперации уровня примеров изображена на рис. 6.13.

6.2. Объекты

Необходимость изображения взаимосвязей не только между множествами объектов (классами) модели, но и между отдельными объектами, реализующими эти классы, может возникнуть в силу самых различных причин. Наиболее важной из них является изображение реализации отдельного варианта использования модели. В ранних версиях языка UML для этой цели служила диаграмма объектов, которая в последующем была заменена канонической диаграммой кооперации. При этом основными элементами или графическими примитивами, из которых образуется остов или каркас диаграммы кооперации уровня примеров, являются объекты.

Объект (object) является отдельным экземпляром класса, который создается на этапе реализации модели (выполнения программы). Он может иметь свое собственное имя и конкретные значения атрибутов. Рассмотрим особенности семантики и графической нотации объектов, из которых строятся диаграммы кооперации на уровне примеров.

Для диаграмм кооперации уровня примеров полное имя объекта в целом представляет собой строку текста, разделенную двоеточием и записанную в формате:

```
<собственное имя объекта> '/'<Имя роли классификатора>
:<Имя классификатора> [','<Имя классификатора>]* .
```

Здесь элементы `Имя роли классификатора` и `Имя классификатора` аналогичны рассмотренным выше, а вся запись имени объекта подчеркивается. Если указано `собственное имя объекта`, то оно должно начинаться со строчной буквы. В этом формате, как правило, имя роли классификатора записывается только при наличии собственного имени объекта на уровне примеров.

В отдельных случаях собственное имя объекта может отсутствовать. Такой объект принято называть *анонимным*, при этом обязательно записывается двоеточие перед именем соответствующего класса (рис. 6.6, *в*). Отсутствовать может и имя класса — такой объект называется *сиротой*. Для него записывается только собственное имя объекта, а двоеточие и имя класса не указываются (рис. 6.6, *г*). Если для объектов указываются атрибуты, то в большинстве случаев они принимают конкретные значения (рис. 6.6, *б*). Для объектов на рис. 6.6, *д*, *е* дополнительно указаны роли, которые они играют в кооперации.

Таким образом, на диаграммах кооперации уровня примеров могут встретиться следующие варианты возможных записей имен объектов:

- o : C — объект с собственным именем o, образуемый на основе класса C;
- : C — анонимный объект, образуемый на основе класса C;
- o : (или просто o) — объект-сирота с собственным именем o;
- o / R : C — объект с собственным именем o, образуемый на основе класса C и играющий роль R;
- / R : C — анонимный объект, образуемый на основе класса C и играющий роль R;
- o / R — объект-сирота с собственным именем o, играющий роль R;
- / R — анонимный объект и одновременно объект-сирота, играющий роль R.

Отдельные примеры изображения объектов на диаграммах кооперации уровня примеров приводятся на рис. 6.6.

Рис. 6.6. Примеры графических изображений объектов на диаграммах кооперации уровня примеров

Примечание

В прямоугольнике объекта диаграммы кооперации уровня примеров имя объекта, имя роли с символом "/" или имя класса могут отсутствовать. Однако двоеточие всегда должно стоять перед именем класса, а косая черточка — перед именем роли. Следует еще раз акцентировать внимание на том обстоятельстве, что применительно к объектам диаграммы кооперации уровня примеров вся запись должна быть подчеркнута, а собственное имя объекта должно быть записано со строчной буквы. Наоборот, на диаграммах кооперации уровня спецификации полные имена объектов не подчеркиваются. Увы, многие разработчики просто не обращают внимания на эти нюансы языка UML.

6.2.1. Мультиобъект

Мультиобъект (multiobject) представляет собой множество объектов, которые могут быть образованы на основе одного класса. На диаграмме кооперации уровня примеров мультиобъект используется для того, чтобы показать операции и сигналы, которые адресованы всему множеству объектов, а не только отдельному объекту. Мультиобъект изображается двумя прямоугольниками, один из которых выступает из-за верхней правой вершины другого (рис. 6.7, *а*). При этом стрелка взаимосвязи относится ко всему множеству объектов, которые обозначает данный мультиобъект. На диаграмме кооперации может быть явно указано отношение агрегации (композиции) между мультиобъектом и отдельным объектом из его множества (рис. 6.7, *б*).

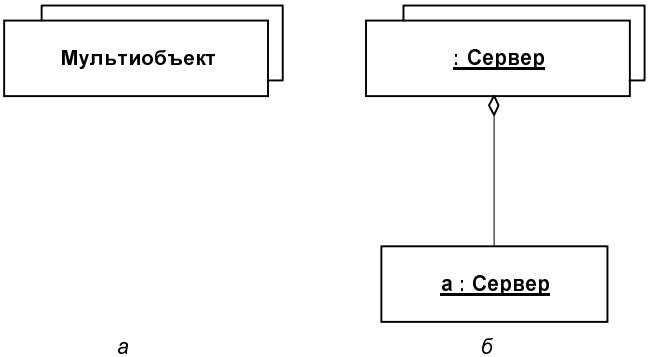


Рис. 6.7. Графическое изображение мультиобъектов на диаграмме кооперации

6.2.2. Активный объект

В контексте языка UML все объекты делятся на две категории: пассивные и активные.

Пассивный объект оперирует только данными и не может инициировать деятельность по управлению другими объектами. Однако пассивные объекты могут посылать сигналы в процессе выполнения запросов, которые они обрабатывают.

Активный объект (active object) имеет свой собственный поток управления (процесс) или нить (thread) и может инициировать деятельность по управлению другими объектами. При этом под нитью понимается некоторый облегченный поток управления (подпроцесс), который может выполняться параллельно с другими вычислительными нитями (подпроцессами) в пределах одного вычислительного процесса или потока управления.

Примечание

Отличие между процессом и нитью условно и заключается в степени использования ресурсов системы. Говоря о процессе, имеют в виду ресурсоемкий поток управления, т. е. такой процесс, который полностью монополизирует ресурсы системы. Нить может использовать лишь небольшую часть ресурсов системы и выполняться в рамках некоторого процесса-владельца. Примером может служить выполнение небольшой программы (утилиты) в адресном пространстве некоторого процесса.

Активные объекты на канонических диаграммах обозначаются прямоугольником с утолщенными границами (рис. 6.8). Иногда, чтобы явно выделить активный объект на диаграмме, может быть дополнительно указано ключевое слово (помеченное значение) `{active}`. Каждый активный объект является владельцем некоторого потока (процесса) управления или нити.

В приведенном фрагменте диаграммы (рис. 6.8) кооперации активный объект `a : Абонент` является инициатором процесса установления соединения (вызывающим абонентом) для обмена информацией с другим абонентом, который на данном фрагменте диаграммы не показан.

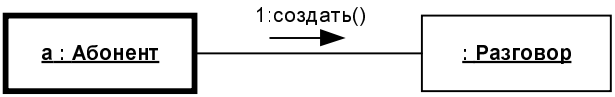


Рис. 6.8. Графическое изображение активного объекта (слева) на диаграмме кооперации

В следующем примере рассматривается ситуация с вызовом функции печати из текстового редактора (рис. 6.9). Анонимный активный объект класса `Текстовый редактор` вначале посылает сообщение анонимному мультиобъекту класса `Принтер`. Это сообщение инициирует выбор единственного объекта класса `Принтер`, возможно, удовлетворяющего некоторым дополнительным условиям. После этого выбранному объекту посылается сообщение о необходимости напечатать документ, загруженный в текстовый редактор.

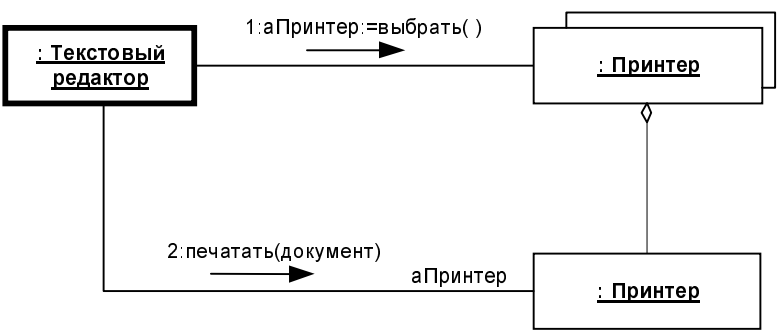


Рис. 6.9. Фрагмент диаграммы кооперации для вызова функции печати из текстового редактора

6.2.3. Составной объект

Составной объект (composite object) или объект-композит предназначен для представления объекта, имеющего собственную структуру и внутренние потоки (нити) управления. Составной объект является экземпляром класса-композита, который связан отношением композиции *(см. главу 5)* со своими частями. Аналогичные отношения связывают между собой и соответствующие объекты.

На диаграммах кооперации такой составной объект изображается как обычный объект, состоящий из двух секций: верхней и нижней. В верхней секции

записывается имя составного объекта, а в нижней — его объекты-части вместо списка его атрибутов (рис. 6.10). При этом допускается иметь в качестве частей другие составные объекты.

Рис. 6.10. Графическое изображение составного объекта на диаграмме кооперации

При изображении диаграммы кооперации уровня примеров следует помнить, что каждый объект представляет собой экземпляр соответствующего класса, а отношения между объектами описываются с помощью связей, которые являются экземплярами соответствующих ассоциаций.

6.3. Связи

Связь (link) является экземпляром или примером произвольной ассоциации. Связь как элемент языка UML может иметь место между двумя и более объектами. Бинарная связь на диаграмме кооперации изображается отрезком сплошной линии, соединяющей два прямоугольника объектов (см. рис. 6.8, 6.9). На каждом из концов этой линии дополнительно могут быть явно указаны имена ролей соответствующей ассоциации.

Связи не имеют собственных имен, поскольку полностью идентичны как экземпляры ассоциации. Другими словами, все связи на диаграмме кооперации могут быть только анонимными и при необходимости записываются без двоеточия перед именем ассоциации. Однако чаще всего имена связей (ассоциаций) на диаграммах кооперации не указываются. Для связей не указывается также и кратность концевых точек. Однако другие обозначения специальных случаев отношений (агрегация, композиция) могут присутствовать на отдельных концах связей. Например, символ связи типа агрегации

между мультиобъектом класса `Принтер` и отдельным объектом класса `Принтер` (рис. 6.9).

6.3.1. Стереотипы связей

Связь может иметь некоторый стереотип, который записывается рядом с одним из ее концов и указывает на особенность реализации данной связи. В языке UML для этой цели могут использоваться следующие стереотипы:

- `<<association>>` — связь-ассоциация (строго говоря, связь, являющаяся экземпляром ассоциации между соответствующими классами). Предполагается по умолчанию, поэтому этот стереотип можно не указывать;
- `<<parameter>>` — параметр операции или метода. Соответствующий объект может быть только параметром некоторой операции;
- `<<local>>` — локальная переменная. Ее область видимости ограничена только соседним объектом;
- `<<global>>` — глобальная переменная. Ее область видимости распространяется на всю диаграмму кооперации;
- `<<self>>` — рефлексивная связь объекта с самим собой, которая допускает передачу объектом сообщения самому себе. На диаграмме кооперации рефлексивная связь изображается петлей в верхней части прямоугольника объекта.

Некоторые примеры связей с различными стереотипами изображены на рис. 6.11. Здесь представлена обобщенная схема некоторой конкретной компании с именем `c`, которая состоит из отделов (анонимный мультиобъект класса `Отдел`). Последние, в свою очередь, состоят из сотрудников (анонимный мультиобъект класса `Сотрудник`). Рефлексивная связь указывает на тот факт, что менеджер отдела является в то же время и его сотрудником.

Примечание

Поскольку на данной диаграмме отсутствуют сообщения, то она не является, строго говоря, законченной диаграммой кооперации уровня примеров. Скорее это специальный случай диаграммы кооперации, который иногда называют диаграммой объектов. В случае N-арной взаимосвязи соответствующая связь на диаграмме кооперации изображается аналогично N-арной ассоциации диаграммы классов с использованием символа ромба.

Как было отмечено выше, цель взаимодействия в контексте языка UML заключается в том, чтобы специфицировать коммуникацию между множеством взаимодействующих объектов. Каждое взаимодействие описывается совокупностью сообщений, которыми участвующие в нем объекты обмениваются между собой.

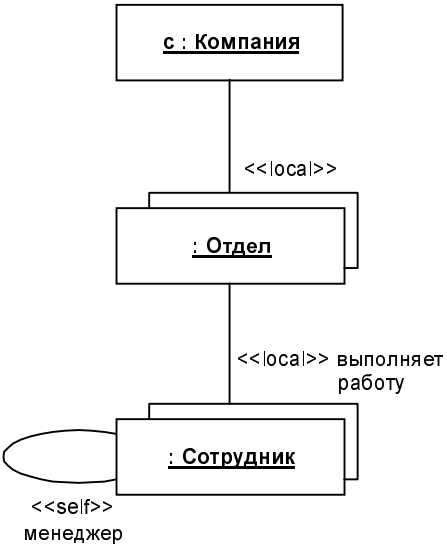


Рис. 6.11. Графическое изображение связей с различными стереотипами

6.4. Сообщения

Сообщение (message) на диаграмме кооперации специфицирует коммуникацию между двумя объектами, один из которых передает другому некоторую информацию. При этом первый объект предполагает, что после получения сообщения вторым объектом последует выполнение некоторого действия. Таким образом, именно сообщение является причиной или стимулом для начала выполнения операций, отправки сигналов, создания и уничтожения отдельных объектов. Связь обеспечивает канал для направленной передачи сообщений между объектами от объекта-источника к объекту-получателю.

В этом смысле сообщение представляет собой законченный фрагмент информации, который отправляется одним объектом другому. При этом прием сообщения может инициировать выполнение определенных действий, направленных на решение отдельной задачи тем объектом, которому это сообщение отправлено.

Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают от принимающего объекта выполнения ожидаемых действий. Сообщения могут инициировать выполнение операций объектом соответствующего класса, а параметры этих операций передаются вместе с сообщением.

В таком контексте каждое сообщение имеет направление от объекта, который инициирует и отправляет сообщение, к объекту, который его получает.

Иногда отправителя сообщения называют клиентом, а получателя — сервером. При этом сообщение от клиента имеет форму запроса некоторого сервиса, а реакция сервера на запрос после получения сообщения может быть связана с выполнением определенных действий или передачи клиенту необходимой информации тоже в форме сообщения.

Сообщения в языке UML также специфицируют роли, которые играют объекты — отправитель и получатель сообщения. Сообщения на диаграмме кооперации изображаются дополнительными стрелками рядом с соответствующей связью или ролью ассоциации. Направление стрелки указывает на получателя сообщения. Внешний вид стрелки сообщения имеет определенный смысл. На диаграммах кооперации может использоваться один из трех типов стрелок для обозначения сообщений (рис. 6.12).

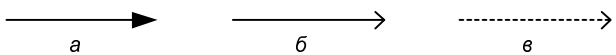


Рис. 6.12. Графическое изображение различных типов сообщений на диаграмме кооперации

1. Сплошная линия с треугольной стрелкой (рис. 6.12, *а*) обозначает вызов процедуры (операции) или передачу потока управления. Сообщения этого типа могут быть использованы параллельно активными объектами, когда один из них передает сообщение этого типа и ожидает, пока не закончится некоторая последовательность действий, выполняемая вторым объектом. Обычно все такие сообщения являются *синхронными*, т. е. инициируются по завершении некоторой деятельности или при выполнении некоторого условия.
2. Сплошная линия с V-образной стрелкой (рис. 6.12, *б*) обозначает асинхронное сообщение в простом потоке управления. В этом случае клиент передает асинхронное сообщение и продолжает выполнять свою деятельность, не ожидая ответа от клиента.
3. Пунктирная линия с V-образной стрелкой (рис. 6.12, *в*) обозначает возврат из вызова процедуры. Стрелки этого типа зачастую отсутствуют на диаграммах кооперации, поскольку неявно предполагается их существование после окончания процесса выполнения некоторой операции или деятельности.

6.4.1. Формат записи сообщений

Каждое сообщение может быть помечено строкой текста, которая имеет следующий формат:

```
<Предшествующие сообщения>
<Выражение последовательности>
```

```
<Возвращаемое значение:=имя сообщения>
<(Список аргументов)>
```

Рассмотрим каждый из этих элементов более подробно.

`Предшествующие сообщения` — это разделенные запятыми номера сообщений, записанные перед наклонной чертой:

```
0  <Номер сообщения','>* <'/'>
```

Если список номеров сообщений пуст, то вся запись, включая наклонную черту (слеш), опускается. Если номера сообщений указываются, то они должны соответствовать номерам других сообщений на этой же диаграмме кооперации.

Смысл указания предшествующих сообщений заключается в том, что данное сообщение не может быть передано, пока не будут переданы своим адресатам все сообщения, номера которых записаны в данном списке.

Пример записи предшествующих сообщений `3` и `4`:

```
0  3,4 / 5: ошибкаЗаписи(сектор)
```

`Выражение последовательности` — это разделенный точками список отдельных термов последовательностей, после которого записывается двоеточие:

```
0  <Терм последовательности'.'…> ':'
```

Каждый из термов представляет отдельный уровень процедурной вложенности в форме законченной итерации. Наиболее верхний уровень соответствует самому левому терму последовательности. Если все потоки управления параллельные, то вложенность отсутствует. Каждый из термов последовательности имеет следующий синтаксис: `[Целое число|Имя] [Рекуррентность]`.

- `Целое число` указывает на порядковый номер сообщения в процедурной последовательности верхнего уровня. Сообщения, номера которых отличаются на единицу, следуют подряд один за другим. Например, сообщение с номером "3.1.4" следует за сообщением с номером "3.1.3" в процедурной последовательности "3.1".

Примечание

Заметим, что сами номера последовательности сообщений с одинаковым префиксом образуют отношение упорядоченности и, соответственно, неявно указывают на предшествующие сообщения. Таким предшествующим сообщением будет сообщение с номером, самая правая цифра которого на единицу меньше, чем у рассматриваемого сообщения. Например, сообщение с номером `3.1.4.6` имеет, в качестве предшествующего, сообщение с номером `3.1.4.5`.

- `Имя` в форме буквы некоторого алфавита используется для спецификации параллельных потоков (нитей) управления. Сообщения, которые отличаются только именем, являются параллельными на этом уровне вложенности.

На одном уровне вложенности все нити управления эквивалентны в смысле приоритета передачи сообщений. Например, сообщения с выражениями "3.1а" и "3.1б" являются параллельными в процедурной последовательности "3.1".

- `Рекуррентность` используется для указания итеративного или условного характера выполнения передачи сообщений. Семантика рекуррентности представляет ноль или больше сообщений, которые должны быть выполнены в зависимости от записанного условия. Возможны два варианта записи рекуррентности:
  - `'*' '['Предложение-итерация']'` для записи итеративного выполнения соответствующего выражения. Итерация представляет последовательность сообщений одного уровня вложенности. Предложение-итерация может быть опущено, если количество итераций никак не специфицируется. Наиболее часто предложение-итерация записывается на некотором псевдокоде или языке программирования. В языке UML формат записи этого предложения не определен. Например, запись рекуррентности "*[*i:=1..n*] " означает последовательную передачу сообщения с параметром *i*, который изменяется от 1 до некоторого целого числа *n* с шагом 1;
  - `'['Предложение-условие']'` для записи ветвления. Эта форма записи специфицирует некоторое условие для данного сообщения, передача которого по данной ветви возможна только при истинности этого условия.

В общем случае предложение-условие (condition-clause) является обычным булевским выражением и предназначено для синхронизации отдельных нитей потока управления. Записывается в квадратных скобках и может быть опущено, если оно отсутствует у данного сообщения. Наличие этого условия обеспечивает передачу сообщения только в том случае, если это условие принимает значение "истина". Предложение-условие может быть записано на обычном тексте, псевдокоде или некотором языке программирования. Например, запись `[x>y]` означает, что сообщение по некоторой ветви будет передано только в том случае, если значение `x` больше значения `y`.

Примечание

Заметим, что предложение-условие записывается так же, как и итерация, но без звездочки. Это можно понимать как некоторую одношаговую итерацию. В общем случае предполагается, что специфицированная итерация выполняется последовательно. Если необходимо отметить возможность параллельного выполнения итерации, то для этой цели в языке UML используется символ "*||". Итерация не распространяется на вложенные уровни данного потока или нити. Каждый уровень должен иметь свое собственное представление для итеративного повторения процедурной последовательности.

Примеры записи предложений-условий без номеров предшествующих сообщений:

```
1.2 [(x>=0)&(x<=255)]: отобразитьНаЭкранеЦвет(x,0,0)
3.1 [количество цифр номера = 7]: набрать_телефонный_номер()
```

`Возвращаемое значение` представляется в форме списка имен значений, возвращаемых после окончания выполнения вызываемой операции объектом-сервером в полной итерации данной процедурной последовательности. Эти идентификаторы могут выступать в качестве аргументов в последующих сообщениях. Если сообщение не возвращает никакого значения, то ни значение, ни оператор присваивания на диаграмме кооперации не указываются.

Например, сообщение

```
1.2.3: p:= найтиДокумент(спецификацияДокумента) ,
```

означает передачу вложенного сообщения с запросом поиска в базе данных нужного документа по его спецификации, причем источнику сообщения должен быть возвращен найденный документ в форме значения `p`.

`Имя сообщения`, записанное в сигнатуре после возвращаемого значения, означает имя события, которое инициируется объектом-получателем сообщения после его приема. Наиболее часто таким событием является вызов операции у объекта-получателя. Это может быть реализовано различными способами, один из которых — явное указание в качестве имени сообщения вызываемой операции. Тогда соответствующая операция должна быть определена в том классе, которому принадлежит объект-получатель.

`Список аргументов` представляет собой разделенные запятыми и заключенные в круглые скобки действительные параметры той операции, вызов которой инициируется данным сообщением. Он может быть пустым, однако скобки все равно записываются. Для записи аргументов также может быть использован некоторый псевдокод или язык программирования.

Так, в приведенном выше примере сообщения

```
1.2.3: p:=найтиДокумент(спецификацияДокумента)
```

`найтиДокумент()` является именем сообщения (операции), а `спецификацияДокумента` — списком аргументов, состоящим из единственного действительного параметра операции. При этом имя сообщения означает обращение к операции `найтиДокумент()`, которая должна быть определена в соответствующем классе объекта-получателя.

6.4.2. Стереотипы сообщений

В языке UML предусмотрены некоторые стандартные действия, выполняемые в ответ на получение соответствующего сообщения. Они могут быть явно указаны на диаграмме кооперации в форме стереотипа перед именем сообщения, к которому они относятся, или выше его. В этом случае они

записываются в угловых кавычках. В языке UML определены следующие стереотипы сообщений:

- `<<call>>` (вызвать) — сообщение, требующее вызова операции или процедуры объекта-получателя. Если сообщение с этим стереотипом рефлексивное, то оно инициирует локальный вызов операции у самого пославшего это сообщение объекта;
- `<<return>>` (возвратить) — сообщение, возвращающее значение выполненной операции или процедуры вызвавшему ее объекту. Значение результата может инициировать ветвление потока управления;
- `<<create>>` (создать) — сообщение, требующее создания другого объекта для выполнения определенных действий. Созданный объект может стать активным (ему передается поток управления), а может остаться пассивным;
- `<<destroy>>` (уничтожить) — сообщение с явным требованием уничтожить соответствующий объект. Посылается в том случае, когда необходимо прекратить нежелательные действия со стороны существующего в системе объекта либо когда объект больше не нужен и должен освободить задействованные им системные ресурсы;
- `<<send>>` (послать) — обозначает посылку другому объекту некоторого сигнала, который асинхронно инициируется одним объектом и принимается (перехватывается) другим. Отличие сигнала от сообщения заключается в том, что сигнал должен быть явно описан в том классе, объект которого инициирует его передачу.

При разработке диаграмм кооперации вначале изображаются объекты и связи между ними (например, рис. 6.5). Далее на диаграмму кооперации уровня примеров необходимо нанести все сообщения, указав их порядок и другие семантические особенности. Окончательный фрагмент диаграммы кооперации модели телефонного разговора изображен на рис. 6.13.

Заметим, что для объекта класса `Коммутатор` указано стандартное помеченное значение `{persistent}`, которое означает, что этот объект создается (существует) в момент начала выполнения основного процесса и уничтожается вместе с ним. Для объекта класса `Разговор` указано стандартное ограничение `{transient}`, которое означает, что этот объект создается в ходе выполнения основного процесса и уничтожается до его завершения. Напомним, что помеченные значения и ограничения являются стандартными элементами механизма расширения языка UML.

Примечание

Изображенная на рис. 6.13 диаграмма кооперации представляет, строго говоря, модель только для начала телефонного разговора. Эта диаграмма может быть дополнена сообщениями, необходимыми для окончания разговора, что читателям предлагается выполнить самостоятельно в качестве упражнения.

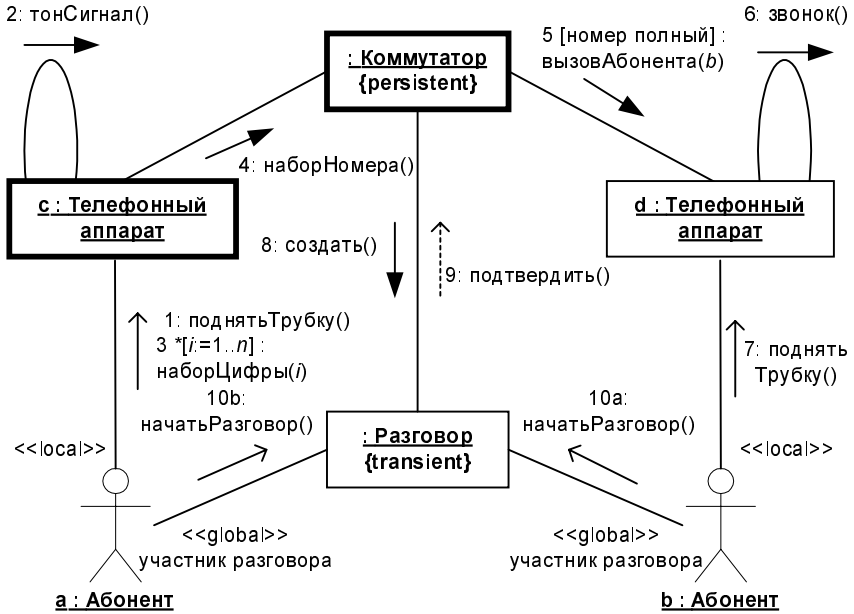


Рис. 6.13. Диаграмма кооперации для представления модели телефонного разговора

Как нетрудно заметить, диаграмма кооперации оказывается необходимым представлением модели и позволяет представить различные типы структурных отношений (ассоциации, композиции, агрегации) между взаимодействующими объектами. При этом диаграмма кооперации, как видно из примера с телефонным разговором, не содержит ни временных особенностей передачи сообщений, ни особенностей жизненного цикла участвующих в данной кооперации объектов. Для представления подобных аспектов поведения моделируемой системы служат диаграммы последовательности, которые рассматриваются в *главе 7*.

6.5. Пример построения диаграммы кооперации системы управления банкоматом

Продолжая построение модели сквозного примера модели системы управления банкоматом, построим диаграмму кооперации уровня примеров для реализации варианта использования `Снятие наличных по кредитной карточке`, описывающей типичный ход событий. Как и диаграмма классов, диаграмма кооперации также может иметь различный вид, что отражает субъективный характер разработчиков на отдельные аспекты реализации

модели. Один из возможных вариантов диаграммы кооперации для данного примера изображен на рис. 6.14.

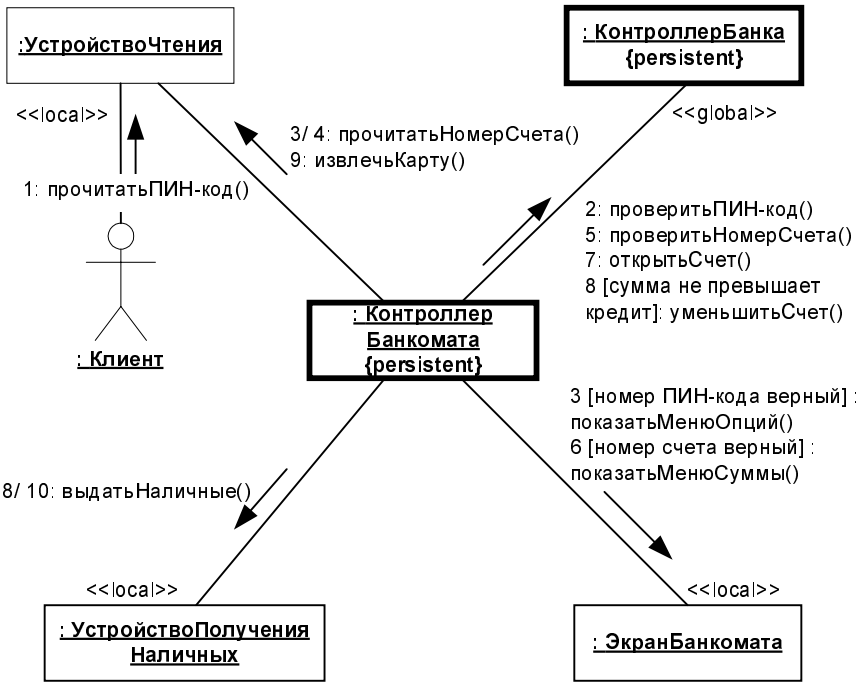


Рис. 6.14. Диаграмма кооперации системы управления банкоматом

Данная диаграмма уровня примеров содержит 5 объектов, два из которых представлены в форме активных объектов, а три — в форме пассивных. При этом все объекты диаграммы являются анонимными. В качестве имен сообщений указаны имена операций, которые специфицированы у соответствующих классов. Предложения-условия сообщений с номерами 3, 6 и 8 записаны на обычном тексте. Эти условия отражают возможность ветвления процесса снятия наличных по кредитной карточке и выполнения исключений сценария соответствующего варианта использования.

Таким образом, изображенная диаграмма кооперации уровня примеров описывает типичный ход событий, приводящий к успеху, только для одного варианта использования разрабатываемой модели системы управления банкоматом. Полная модель системы должна содержать дополнительные диаграммы кооперации, описывающие реализации всех специфицированных вариантов использования (типичный ход событий и все исключения). Все эти диаграммы кооперации предлагается построить читателям самостоятельно в качестве упражнения.

6.6. Заключительные рекомендации по построению диаграмм кооперации

Построение диаграммы кооперации можно начинать либо сразу после построения диаграммы вариантов использования, либо после построения диаграммы классов. В первом случае каждый из вариантов использования может быть специфицирован в виде отдельной диаграммы кооперации уровня спецификации. Такая диаграмма способствует более полному пониманию особенностей реализации отдельных функциональных требований к разрабатываемой системе, хотя и не может содержать всю информацию, необходимую для их реализации. При этом множества объектов диаграммы кооперации специфицируются именами ролей классификаторов, каждая из которых служит прототипом некоторого класса.

Однако наиболее часто диаграмма кооперации строится после разработки диаграммы классов. В любом случае, после построения диаграммы классов, каждая из диаграмм кооперации уровня спецификации может уточняться в виде соответствующей диаграммы уровня примеров. Важно понимать, что диаграмма кооперации этого уровня может содержать те и только те объекты и связи, которые уже определены на построенной ранее диаграмме классов. В противном случае, если возникает необходимость включения в диаграмму кооперации объектов, которые создаются на основе отсутствующих классов, то диаграмма классов должна быть модифицирована посредством включения в нее явного описания этих классов.

В отличие от диаграммы последовательности, на диаграмме кооперации изображаются только такие отношения между объектами, которые играют роль информационных каналов при взаимодействии. С другой стороны, на этой диаграмме не указывается время в виде отдельного измерения. Последовательность взаимодействий объектов и параллельные потоки сообщений могут быть определены лишь с помощью порядковых номеров сообщений. Следовательно, если необходимо явно специфицировать взаимосвязи между объектами в реальном времени, лучше это делать на диаграмме последовательности.

Следует помнить, что на диаграмме кооперации изображаются только те объекты, которые непосредственно в ней участвуют. При этом объекты могут выступать в различных ролях, которые должны быть явно указаны на соответствующих концах связей диаграммы. Применение стереотипов унифицирует кооперацию, обеспечивая ее адекватную интерпретацию, как со стороны заказчиков, так и со стороны разработчиков. Тем не менее, целесообразно различать терминологию, используемую на диаграммах кооперации уровня спецификации и уровня примеров.

Так, при построении диаграмм кооперации уровня спецификации желательно применять наиболее понятную заказчику терминологию, избегая

технических фраз и словосочетаний. Например, "оформить заказ", "отгрузить товар", "представить отчет", "разработать план" и т. д. Такие известные разработчикам слова как "сервер", "защищенный протокол", "закрытая операция класса", а также стереотипы и помеченные значения на этом уровне применять не рекомендуется. На уровне спецификации нужно стремиться достичь по возможности полного взаимопонимания между заказчиком и командой разработчиков особенностей реализации ключевых вариантов использования проектируемой системы в контексте их кооперации.

При построении диаграмм кооперации уровня примеров терминология может более точно отражать все аспекты реализации соответствующих объектов и связей. Поскольку диаграмма этого уровня является документацией для разработчиков системы, здесь допустимо использовать весь арсенал стереотипов, ограничений и помеченных значений, который имеется в языке UML. Если типовых обозначений недостаточно, разработчики могут дополнить диаграмму собственными элементами, используя механизм расширений языка UML.

Процесс построения диаграммы кооперации уровня примеров должен быть согласован с процессами построения диаграммы классов и диаграммы последовательности. В первом случае, как уже отмечалось, необходимо следить за использованием только тех объектов, для которых определены порождающие их классы. Во втором случае необходимо согласовывать последовательности передаваемых сообщений. Речь идет о том, что не допускается различный порядок следования сообщений для моделирования одного и того же взаимодействия на диаграмме кооперации и диаграмме последовательности.

Таким образом, диаграмма кооперации, с одной стороны, обеспечивает концептуально согласованный переход от статической модели диаграммы классов к динамическим моделям поведения, представляемым диаграммами последовательности, состояний и деятельности. С другой стороны, диаграмма этого типа предопределяет особенности реализации модели на диаграммах компонентов и развертывания, которые являются предметом рассмотрения в двух последующих главах книги.

В заключение приводится сводка основных графических обозначений, которые могут использоваться при построении диаграмм кооперации (табл. 6.1).

Таблица 6.1. *Основные графические элементы диаграмм кооперации*

Графическое изображение	Название
имя объекта: Имя класса	Объект (object)

Таблица 6.1 *(окончание)*

Графическое изображение	Название
	Мультиобъект (multiobject)
	Активный объект (active object)
	Связь (link)
	Рефлексивная связь
	Вызов процедуры (операции)
	Асинхронное сообщение
	Возврат из вызова процедуры

Глава 7

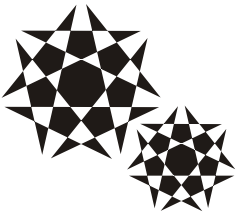


Диаграмма последовательности (sequence diagram)

Как отмечалось в *главе 6*, особенности взаимодействия элементов моделируемой системы могут быть представлены на диаграммах кооперации и последовательности. Хотя диаграммы кооперации и используются для спецификации динамики поведения систем, но время в явном виде в них отсутствует. Однако временной аспект поведения может иметь существенное значение при моделировании синхронных процессов, описывающих взаимодействия объектов. Именно для этой цели в языке UML используются диаграммы последовательности, которые и являются предметом изучения настоящей главы.

С помощью диаграммы последовательности можно описать полный контекст взаимодействий как своеобразный временной график "жизни" всей совокупности объектов, взаимодействующих между собой для реализации варианта использования программной системы, достижения бизнес-цели или выполнения какой-либо задачи.

7.1. Объекты

Отдельные аспекты спецификации объектов как элементов диаграмм кооперации уровня примеров уже рассматривались ранее *(см. главу 6)*. На диаграмме последовательности также изображаются объекты, которые непосредственно участвуют во взаимодействии, при этом никакие статические связи с другими объектами не визуализируются. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени. При этом диаграмма последовательности имеет как бы два измерения. Одно — слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии.

При этом каждый объект графически изображается в форме прямоугольника и располагается в верхней части своей линии жизни (рис. 7.1). Внутри прямоугольника записываются собственное имя объекта со строчной буквы и имя класса, разделенные двоеточием. При этом вся запись подчеркивается,

что является признаком объекта, который, как указывалось ранее, представляет собой экземпляр класса.

Примечание

Для объектов диаграммы последовательности остаются справедливыми правила именования, рассмотренные ранее применительно к диаграммам кооперации уровня примеров. Тем самым не исключается ситуация, когда на диаграмме последовательности может отсутствовать собственное имя объекта, при этом должно быть указано имя класса. Такой объект считается анонимным. Может отсутствовать и имя класса, но при этом должно быть указано собственное имя объекта. Такой объект считается сиротой. Что касается ролей классификаторов, то в именах объектов на диаграммах последовательности они, как правило, не указываются.

Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия (`объект A` на рис. 7.1). Правее изображается другой объект, который непосредственно взаимодействует с первым. Таким образом, порядок расположения объектов на диаграмме последовательности определяется исключительно соображениями удобства визуализации их взаимодействия друг с другом.

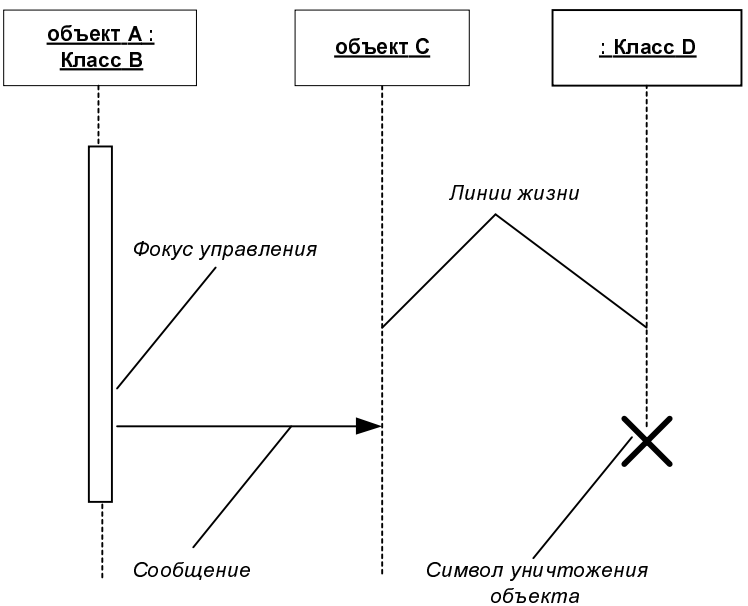


Рис. 7.1. Графические элементы диаграммы последовательности

Второе измерение диаграммы последовательности — вертикальная временная ось, направленная сверху вниз. Начальному моменту соответствует самая

верхняя часть диаграммы. При этом процесс взаимодействия объектов реализуется посредством сообщений, которые посылаются одними объектами другим. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и образуют некоторый порядок относительно времени своей инициализации. Другими словами, сообщения, расположенные на диаграмме последовательности выше, передаются раньше тех, которые расположены ниже. При этом масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа "раньше-позже".

7.1.1. Линия жизни объекта

Линия жизни объекта (object lifeline) изображается пунктирной вертикальной линией, ассоциированной с единственным объектом на диаграмме последовательности. Линия жизни служит для обозначения периода, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней (`объект A` и `объект C` на рис. 7.1).

С другой стороны, отдельные объекты, выполнив свою роль в системе, могут быть уничтожены (разрушены), чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML

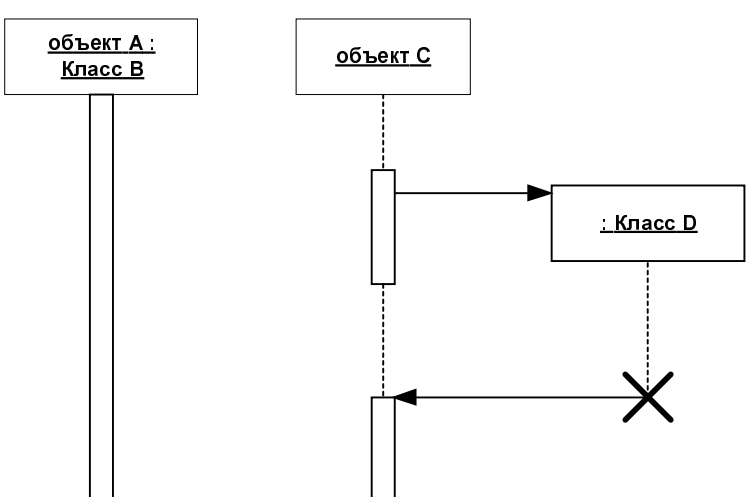


Рис. 7.2. Графическое изображение различных вариантов линий жизни и фокусов управления объектов

используется специальный символ в форме латинской буквы "X" (анонимный объект, образованный от класса D на рис. 7.1, 7.2). Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет и этот объект должен быть исключен из всех последующих взаимодействий.

Вовсе не обязательно создавать все объекты в начальный момент. Отдельные объекты в системе могут создаваться по мере необходимости, существенно экономя ресурсы системы и повышая ее производительность. В этом случае прямоугольник такого объекта изображается не в верхней части диаграммы последовательности, а в той ее части, которая соответствует моменту создания объекта (анонимный объект, образованный от класса D на рис. 7.2). При этом прямоугольник объекта вертикально располагается в том месте диаграммы, которое по оси времени совпадает с моментом его возникновения в системе. Очевидно, объект обязательно создается со своей линией жизни и, возможно, с фокусом управления.

7.1.2. Фокус управления

В процессе функционирования объектно-ориентированных систем одни объекты могут находиться в активном состоянии, непосредственно выполняя определенные действия или в состоянии пассивного ожидания сообщений от других объектов. Чтобы явно выделить подобную активность объектов, на диаграммах последовательности применяется специальное понятие, получившее название *фокуса управления* (focus of control). Фокус управления изображается в форме вытянутого узкого прямоугольника (`объект A` на рис. 7.1), верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности), а ее нижняя сторона — окончание фокуса управления (окончание активности). Этот прямоугольник располагается ниже обозначения соответствующего объекта и может заменять его линию жизни (`объект A` на рис. 7.2), если на всем ее протяжении он является активным.

С другой стороны, периоды активности объекта могут чередоваться с периодами его пассивности или ожидания. В этом случае у такого объекта фокусы управления изменяют свое изображение на линию жизни и наоборот (`объект C` на рис. 7.2). Важно понимать, что получить фокус управления может только существующий объект, у которого в этот момент имеется линия жизни. Если же некоторый объект был уничтожен, то вновь возникнуть в системе он уже не может. Вместо него может быть создан лишь экземпляр этого же класса, который, строго говоря, будет являться другим объектом.

В отдельных случаях инициатором взаимодействия в системе может быть актер или внешний пользователь. В этом случае актер изображается на диаграмме последовательности самым первым объектом слева со своим фокусом управления (рис. 7.3). Наиболее часто актер и его фокус управления бу-

дут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. При этом сам актер может иметь собственное имя либо оставаться анонимным.

В отдельных случаях объект может посылать сообщения самому себе, инициируя так называемые *рефлексивные* сообщения. Для этой цели служит специальное изображение (`объект A` на рис. 7.3). Такие сообщения изображаются прямоугольником со стрелкой, начало и конец которой совпадают. Подобные ситуации возникают, например, при обработке нажатий клавиш клавиатуры при вводе текста в редактируемый документ, при наборе цифр номера телефона абонента.

Если в результате рефлексивного сообщения создается новый подпроцесс или нить управления, то говорят о рекурсивном или вложенном фокусе управления. На диаграмме последовательности *рекурсия* обозначается небольшим прямоугольником, присоединенным к правой стороне фокуса управления того объекта, для которого изображается данное рекурсивное взаимодействие (анонимный объект, полученный от класса C на рис. 7.3).

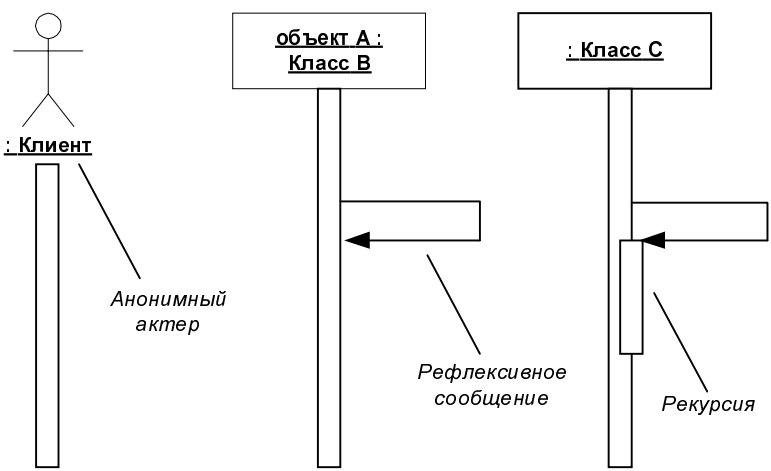


Рис. 7.3. Графическое изображение актера, рефлексивного сообщения и рекурсии на диаграмме последовательности

7.2. Сообщения на диаграмме последовательности

Сообщения как элементы языка UML уже рассматривались ранее при изучении диаграммы кооперации *(см. главу 6)*. Хотя изображения стрелок сообщений аналогичны рассмотренным ранее, применительно к диаграммам последовательности сообщения имеют некоторые дополнительные семантические

особенности. При этом на диаграмме последовательности все сообщения упорядочены по времени своей передачи в моделируемой системе, хотя номера у них могут не указываться.

На диаграммах последовательности могут присутствовать три разновидности сообщений, каждое из которых имеет свое графическое изображение (рис. 7.4).

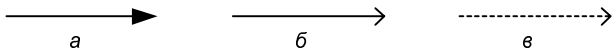


Рис. 7.4. Графическое изображение различных видов сообщений между объектами на диаграмме последовательности

1. Первая разновидность сообщения (рис. 7.4, *а*) является наиболее распространенной и используется для вызова процедур, выполнения операций или обозначения отдельных вложенных потоков управления. Начало этой стрелки, как правило, соприкасается с фокусом управления того объекта-клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия. При этом принимающий объект может получить фокус управления, становясь в этом случае активным. Передающий объект может потерять фокус управления или остаться активным.
2. Вторая разновидность сообщения (рис. 7.4, *б*) используется для обозначения простого асинхронного сообщения, которое передается в произвольный момент. Передача такого сообщения обычно не сопровождается получением фокуса управления объектом-получателем.
3. Третья разновидность сообщения (рис. 7.4, *в*) используется для возврата из вызова процедуры. Примером может служить простое сообщение о завершении некоторых вычислений без предоставления результата расчетов объекту-клиенту. В процедурных потоках управления эта стрелка может быть опущена, поскольку ее наличие неявно предполагается в конце активизации объекта. В то же время считается, что каждый вызов процедуры имеет свою пару — возврат вызова. Для непроцедурных потоков управления, включая параллельные и асинхронные сообщения, стрелка возврата должна указываться явным образом.

Обычно сообщения изображаются горизонтальными стрелками, соединяющими линии жизни или фокусы управления двух объектов на диаграмме последовательности. При этом неявно предполагается, что время передачи сообщения достаточно мало по сравнению с процессами выполнения действий объектами. Считается также, что за время передачи сообщения с соответствующими объектами не может произойти никаких событий. Другими словами, состояния объектов остаются без изменения. Если же это предположение не может быть признано справедливым, то стрелка сообщения изо-

бражается под некоторым наклоном, так чтобы конец стрелки располагался ниже ее начала.

Таким образом, в языке UML каждое сообщение ассоциируется с некоторым действием, которое должно быть выполнено принявшим его объектом. При этом действие может иметь некоторые аргументы или параметры, в зависимости от конкретных значений которых может быть получен различный результат. Соответствующие параметры будет иметь и вызывающее это действие сообщение. Более того, значения параметров отдельных сообщений могут содержать условные выражения, образуя ветвление или альтернативные пути основного потока управления.

Примечание

Хотя, в общем случае, взаимодействия на диаграммах последовательности описываются в форме сообщений, применительно к конкретным моделям систем каждое сообщение должно соотноситься с некоторой операцией класса, к которому относится объект-получатель. Именно по этой причине на диаграммах последовательности рядом с сообщениями указываются имена соответствующих операций.

7.2.1. Ветвление потока управления

Одной из особенностей диаграммы последовательности является возможность визуализировать простое ветвление процесса. Для изображения ветвления изображаются две или более стрелки, выходящие из одной точки фокуса управления объекта (`объект A` на рис. 7.5). При этом рядом с каждой

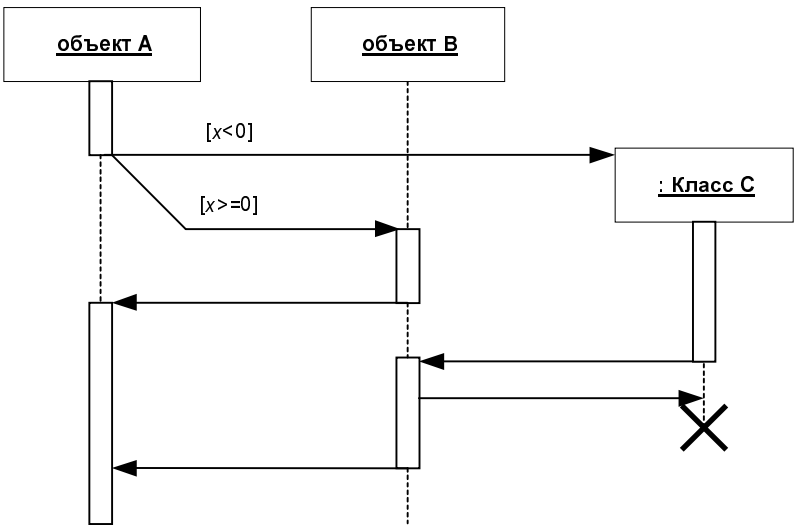


Рис. 7.5. Графическое изображение бинарного ветвления потока управления на диаграмме последовательности

из стрелок должно быть явно указано соответствующее условие ветви в форме булевского выражения. Хотя количество ветвей может быть произвольным, следует помнить, что наличие ветвлений может существенно усложнить интерпретацию диаграммы последовательности. Напомним, что предложение-условие может быть записано в форме обычного текста, псевдокода или выражения языка программирования и должно представлять собой некоторое булевское выражение. Запись этих условий должна исключать одновременную передачу альтернативных сообщений по двум и более ветвям. В этом случае принято говорить об отсутствии конфликта у условий ветвления.

С помощью ветвления можно изобразить и более сложную логику взаимодействия объектов между собой (`объект A` на рис. 7.6). Если условий более двух, то для каждого из них необходимо предусмотреть ситуацию единственного выполнения. Рассматриваемый фрагмент относится к моделированию взаимодействия программной системы обслуживания клиентов в банке. В этом примере диаграммы последовательности `объект A` вызывает выполнение некоторых действий у одного из трех других объектов.

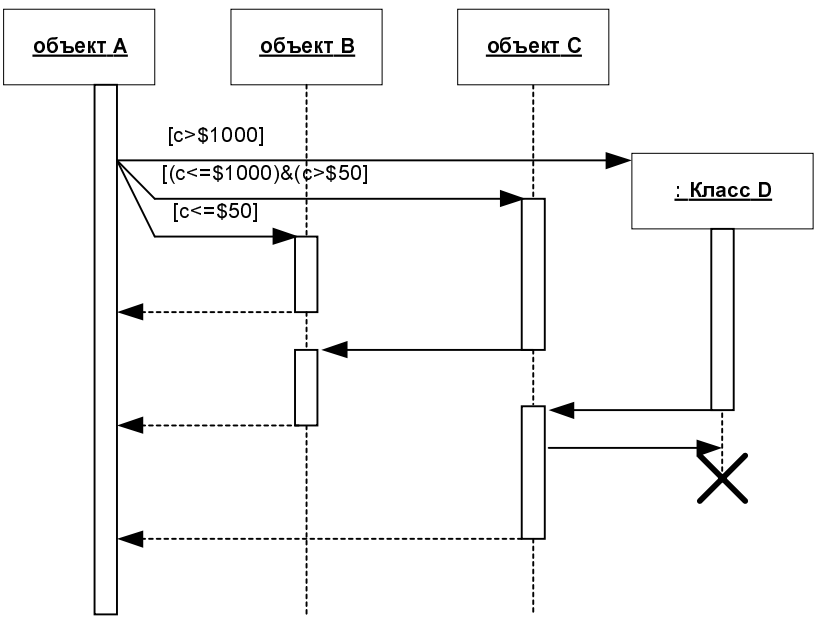


Рис. 7.6. Графическое изображение тернарного ветвления потока управления на диаграмме последовательности

Условием ветвления может служить сумма снимаемых клиентом средств со своего текущего счета. Если эта сумма превышает 1000$, то могут потребоваться дополнительные действия, связанные с созданием и последующим

разрушением объекта, создаваемого от класса D. Если же сумма превышает 50$, но не превышает 1000$, то вызывается некоторая ресурсоемкая операция или процедура объекта C. И, наконец, если сумма не превышает 50$, то вызывается некоторая операция или процедура объекта B. При этом объекты A, B и C постоянно существуют в системе. Последний объект создается от класса D, только если справедливо первое из альтернативных условий. В противном случае он может быть никогда не создан. После выполнения требуемых действий объекты B и C сообщают объекту A об окончании выполнения соответствующих процедур. В свою очередь объект, созданный от класса D, после завершения своих действий уничтожается, передавая управление объекту C, и делает его активным до момента возврата из вызванной процедуры. Объект A имеет постоянный фокус управления, а все остальные объекты получают фокус управления только для выполнения ими соответствующих операций.

На диаграммах последовательности при записи сообщений также могут использоваться стереотипы, рассмотренные ранее при построении диаграммы кооперации *(см. главу 6)*. Их семантика и синтаксис остаются без изменения, как они определены в нотации языка UML. Ниже представлена диаграмма последовательности для рассмотренного случая ветвления, дополненная стереотипными значениями (рис. 7.7). Очевидно, эта диаграмма последовательности является более выразительной и простой для своей интерпретации.

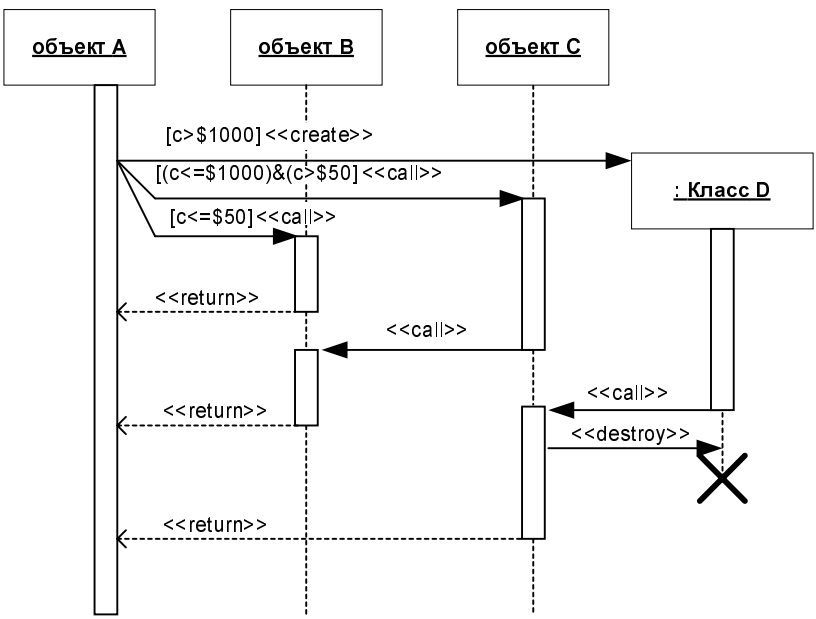


Рис. 7.7. Диаграмма последовательности со стереотипными значениями сообщений

Как уже отмечалось ранее, сообщения могут иметь собственное имя, в качестве которого выступает имя операции, вызов которой инициируют эти сообщения у принимающего объекта. В этом случае рядом со стрелкой записывается имя операции с круглыми скобками, в которых могут указываться параметры или аргументы соответствующей операции. Если параметры отсутствуют, то скобки все равно должны быть изображены после имени операции. Примерами таких операций могут служить следующие: `выдатьКлиентуНаличнымиСумму(n)`, `установитьСоединениеМеждуАбонентами(a,b)`, `сделатьВводимыйТекстНевидимым()`, `податьЗвуковойСигналТревоги()`.

Примечание

Согласно принятой в языке UML системе обозначений такие имена операций записываются на английском языке со строчной буквы и одним словом, возможно состоящим из нескольких сокращенных слов, написанных без пробела и без кавычек. Если нет никаких дополнительных ограничений со стороны инструментальных средств визуализации канонических диаграмм, то это уже дело вкуса отечественного разработчика, какие обозначения ему использовать в русскоязычной транслитерации. Возможно, для этой цели некоторые разработчики предпочтут вариант с подчеркиванием, исключающий пробелы в имени операции: `сделать_вводимый_текст_невидимым()`, чем вариант с заглавными буквами в середине имени операции.

7.2.2. Временные ограничения на диаграммах последовательности

В отдельных случаях выполнение тех или иных действий на диаграмме последовательности может потребовать явной спецификации *временных ограничений*, накладываемых на интервал выполнения операций или передачу сообщений. В языке UML для записи временных ограничений используются фигурные скобки. Временные ограничения могут относиться как к выполнению определенных действий объектами, так и к самим сообщениям, явно специфицируя условия их передачи или приема. Важно понимать, что в отличие от условий ветвления, которые должны выполняться альтернативно, временные ограничения имеют обязательный или директивный характер для ассоциированных с ними элементов модели.

Временные ограничения могут записываться рядом с началом стрелки соответствующего сообщения. Но наиболее часто они записываются слева от этой стрелки на одном уровне с ней. Если временное ограничение относится к конкретному объекту, то имя этого объекта записывается перед именем атрибута и отделяется от нее точкой.

Примерами таких ограничений на диаграмме последовательности могут служить ситуации, когда необходимо явно специфицировать время, в течение

которого допускается передача сообщения от клиента к серверу или обработка запроса клиента сервером:

- {времяПриемаСообщения — времяОтправкиСообщения < 1 сек.}
- {времяОжиданияОтвета < 5 сек.}
- {времяПередачиПакета < 10 сек.}
- {объект А. времяЗвучанияСигналаТревоги = 10 мин.}

В первом из рассмотренных случаев знак "-" (минус) во временном ограничении обозначает арифметическую операцию вычитания. Другие знаки являются обычными знаками сравнения величин. В последнем случае перед временной характеристикой указано имя объекта, к которому она относится.

Примечание

Для записи ограничений, которые являются одним из трех механизмов расширения элементов языка UML, может быть использован естественный язык, псевдокод или выражения языка объектных ограничений. В рассмотренном выше случае использован псевдокод. Наиболее удобно временные ограничения помещать внутри символа примечания. Примечания или комментарии уже рассматривались ранее при изучении других типов диаграмм. Напомним, что для примечаний используется стандартное обозначение — прямоугольник с "заломленным" правым верхним углом. Внутри этого прямоугольника записывается текст примечания на естественном языке. Примечания могут присутствовать на всех типах канонических диаграмм, включая и диаграммы последовательности. При этом каждое из примечаний должно ассоциироваться (присоединяться) с отдельными объектами или сообщениями.

7.3. Пример построения диаграммы последовательности

В качестве примера рассмотрим построение диаграммы последовательности для моделирования процесса телефонного разговора с использованием обычной телефонной сети. Объектами в этом примере являются: два абонента *a* и *b*, два телефонных аппарата *c* и *d*, коммутатор и сам разговор как объект моделирования. При этом как коммутатор, так и разговор являются анонимными объектами.

На первом этапе располагаем выбранные объекты на предполагаемой диаграмме (рис. 7.8). Заметим, что абонентов мы будем рассматривать как актеров, причем первый из них — *a* — играет активную роль, а второй — *b* — пассивную роль. Поэтому первый получает фокус управления сразу после своего появления в системе, а второй имеет только линию жизни. Коммутатор также имеет постоянную активность, что изображается его фокусом управления. Разговор как объект появляется только после установки соединения

и уничтожается с его прекращением. Поэтому он будет изображен позже на этой же диаграмме последовательности.

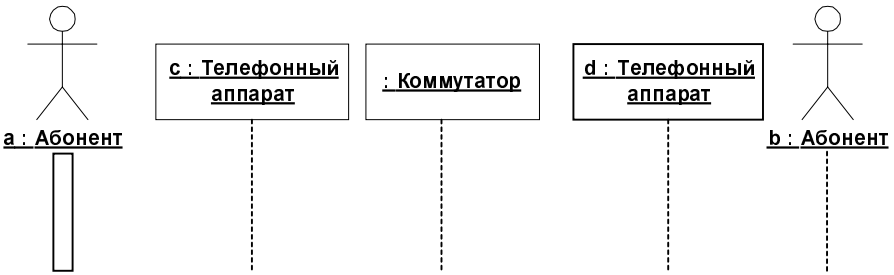


Рис. 7.8. Начальный фрагмент диаграммы последовательности для моделирования телефонного разговора

Процесс взаимодействия в этой системе начинается с поднятия трубки телефонного аппарата первым абонентом. Тем самым он посылает сообщение телефонному аппарату *c*, которое переводит этот аппарат в активное состояние и вызывает следующее действие — подачу тонового сигнала, который будет слышен в телефонной трубке первого абонента. Следующее действие также инициируется первым абонентом — набор цифр телефонного номера. Это представлено в форме итеративного сообщения со знаком "*" (звездочка) слева от его имени, где натуральное *n* обозначает количество цифр набираемого номера.

Заметим, что поднятие телефонной трубки и набор цифр номера являются физическими действиями и поэтому изображаются в форме простых асинхронных сообщений. После набора цифр номера телефона аппарат *c* рекурсивно вызывает процедуру посылки коммутационных импульсов на коммутатор. Последний инициирует вызов абонента *b*, а после поднятия им телефонной трубки коммутатор создает новый объект в моделируемой системе — телефонный разговор. Дополненный фрагмент диаграммы последовательности изображен на следующем рисунке (рис. 7.9).

После создания анонимного объекта от класса `Разговор` абоненты могут начать обмен информацией. При этом длительность разговора в модели никак не ограничивается по времени. В конце концов, обмен информацией заканчивается, и один из абонентов (либо сразу оба) может инициировать окончание разговора (асинхронное действие). После подтверждения окончания разговора абоненты опускают трубки, и разговор заканчивается. Тем самым уничтожается анонимный объект от класса `Разговор`. Окончательный вариант диаграммы последовательности изображен на следующем рисунке (рис. 7.10). Имена отдельных сообщений соответствуют рассмотренным действиям.

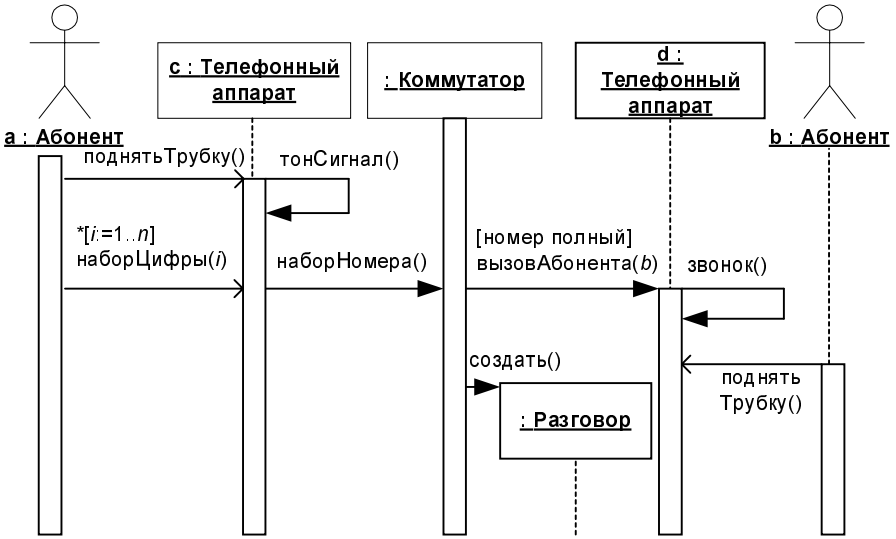


Рис. 7.9. Дополненный фрагмент диаграммы последовательности для моделирования телефонного разговора

Сравнение построенной диаграммы последовательности с рассмотренной ранее *(см. главу 6)* диаграммой кооперации показывает, что из диаграммы последовательности (рис. 7.10) сразу видно время жизни и активности всех объектов. В дополнение к этому диаграммы последовательности могут иметь временные ограничения на передачу отдельных сообщений. Все это является важными особенностями канонических диаграмм этого типа, которые следует учитывать при разработке конкретных моделей систем.

Примечание

Не секрет, что ведение бесконечных телефонных разговоров, особенно в коммунальных условиях проживания, могут вывести из равновесия самых терпеливых из наших читателей. Они правильно заметят, что рассмотренную модель следует дополнить временными ограничениями на существование анонимного объекта от класса `Разговор`. С этой целью можно пометить сообщение `создать()` временной меткой *x*, а сообщение `уничтожить()` — временной меткой *y*. В этом случае временное ограничение может быть записано, например, в следующем виде: `{y – x <10 мин.}`. Что касается других ограничений наряду с предложениями по их технической реализации, то все это предлагается выполнить читателям самостоятельно в качестве упражнения.

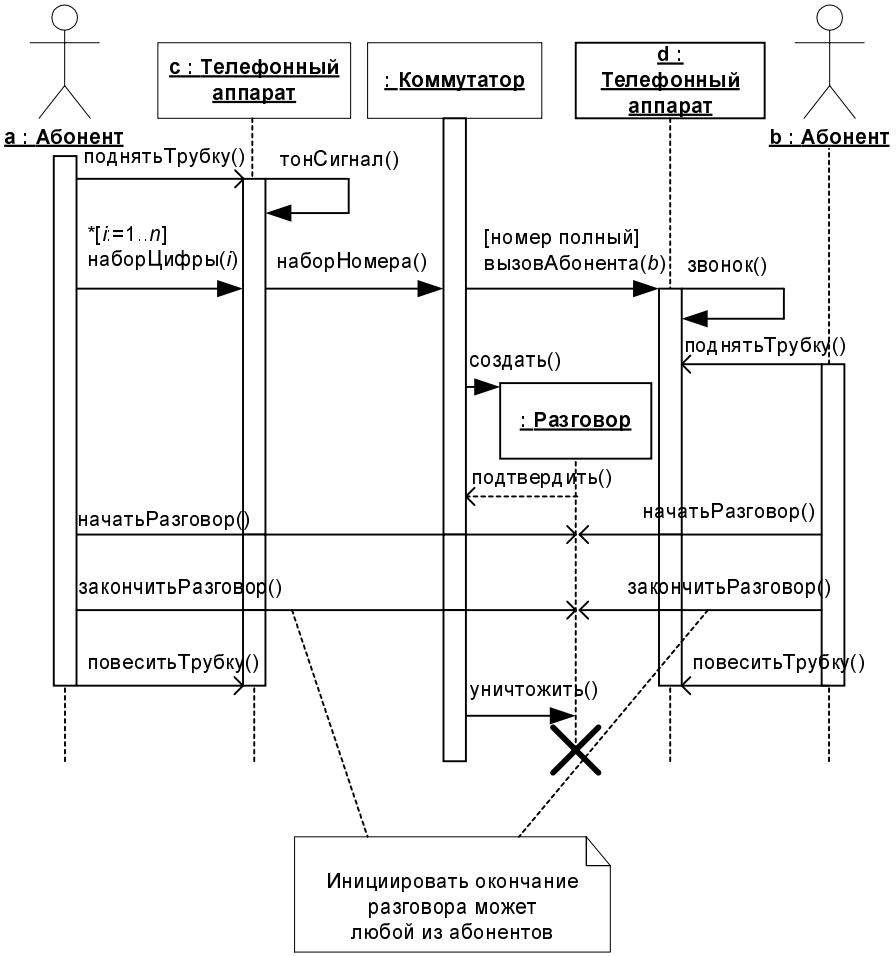


Рис. 7.10. Окончательный вариант диаграммы последовательности для моделирования телефонного разговора

7.4. Пример построения диаграммы последовательности системы управления банкоматом

Продолжая построение сквозного примера модели системы управления банкоматом, построим диаграмму последовательности для реализации варианта использования `Снятие наличных по кредитной карточке`, описывающей типичный ход событий. Вариант диаграммы последовательности, который соответствует построенной ранее диаграмме кооперации для данного примера, изображен на рис. 7.11.

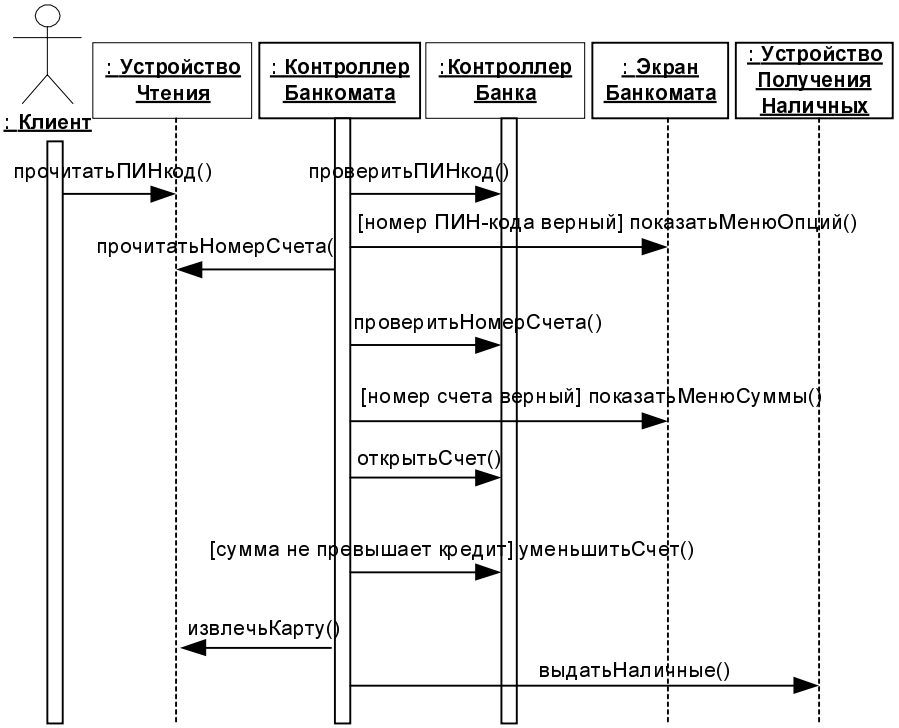


Рис. 7.11. Диаграмма последовательности системы управления банкоматом

Данная диаграмма содержит 5 объектов и актера, все они являются анонимными. Два объекта являются активными, что показано с помощью соответствующих фокусов управления. В качестве имен сообщений также указаны имена операций, которые специфицированы у соответствующих классов. Предложения-условия некоторых сообщений записаны обычным текстом. Эти условия отражают возможность ветвления процесса снятия наличных по кредитной карточке и выполнения исключений сценария соответствующего варианта использования, однако другие ветви на данной диаграмме не показаны.

Таким образом, изображенная диаграмма последовательности описывает типичный ход событий, приводящий к успеху, только для одного варианта использования разрабатываемой модели системы управления банкоматом. Полная модель системы может содержать либо дополненную данную диаграмму последовательности, описывающую реализации всех специфицированных вариантов использования (типичный ход событий и все исключения), либо другие диаграммы последовательности. Все эти диаграммы последовательности предлагается построить читателям самостоятельно в качестве упражнения.

7.5. Заключительные рекомендации по построению диаграмм последовательности

Как уже отмечалось, построение диаграммы последовательности целесообразно начинать с выделения из всей совокупности тех классов, объекты которых участвуют в моделируемом взаимодействии. После этого все объекты наносятся на диаграмму, с соблюдением некоторого порядка инициализации сообщений. Здесь необходимо установить, какие объекты будут существовать постоянно, а какие временно — только на период выполнения требуемых действий.

Когда объекты визуализированы, можно приступать к спецификации сообщений. При этом необходимо учитывать те роли, которые играют сообщения в системе. При необходимости уточнения этих ролей следует использовать их разновидности и стереотипы. Для уничтожения объектов, которые создаются на время выполнения своих действий, нужно предусмотреть явное сообщение.

Наиболее простые случаи ветвления процесса взаимодействия можно изобразить на одной диаграмме с использованием соответствующих графических примитивов. Однако следует помнить, что каждый альтернативный поток управления может существенно затруднить понимание построенной модели. Поэтому общим правилом является визуализация каждого потока управления на отдельной диаграмме последовательности. В этой ситуации такие отдельные диаграммы должны рассматриваться совместно как одна модель взаимодействия.

Дальнейшая детализация диаграммы последовательности связана с введением временных ограничений на выполнение отдельных действий в системе. Для простых асинхронных сообщений временные ограничения могут отсутствовать. Однако необходимость синхронизировать сложные потоки управления, как правило, требуют введение в модель таких ограничений. Общая их запись должна следовать семантике языка объектных ограничений OCL, который рассмотрен в приложении.

В заключение приводится сводка основных графических обозначений, которые могут быть использованы при построении диаграмм последовательности (табл. 7.1).

Таблица 7.1. *Основные графические элементы диаграмм последовательности*

Графическое изображение	Название
имя объекта: Имя класса	Объект (object)

Таблица 7.1 *(окончание)*

Графическое изображение	Название
	Линия жизни (lifeline)
	Фокус управления (focus of control)
	Рекурсия (recursion)
	Рефлексивное сообщение
	Вызов процедуры (операции)
	Асинхронное сообщение
	Возврат из вызова процедуры
	Символ ветвления потока управления

Глава 8

Диаграмма состояний (statechart diagram)

Как было отмечено в первой части книги, одной из характерных особенностей систем различной природы и назначения является взаимодействие между собой отдельных элементов, из которых образованы эти системы. Для представления динамических особенностей взаимодействия элементов модели в контексте реализации вариантов использования предназначены диаграммы кооперации и последовательности. Однако для большинства сложных систем, особенно физических систем реального времени, этих представлений может оказаться недостаточно для моделирования процессов функционирования таких систем как в целом, так и отдельных их подсистем.

Рассмотрим простой пример. Любое техническое устройство, такое как телевизор, компьютер, автомобиль, телефонный аппарат в самом общем случае может характеризоваться такими своими состояниями, как "исправен" и "неисправен". Интуитивно ясно, какой смысл вкладывается в каждое из этих понятий. Более того, использование по назначению данного устройства возможно только тогда, когда оно находится в исправном состоянии. В противном случае необходимо предпринять совершенно конкретные действия по его ремонту и восстановлению работоспособности.

Однако понимание семантики понятия состояния представляет определенные трудности. Дело в том, что характеристика состояний системы не зависит (или слабо зависит) от логической структуры, зафиксированной на диаграмме классов. Поэтому при рассмотрении состояний системы приходится на время отвлечься от особенностей ее объектной структуры и мыслить совершенно другими категориями, описывающими динамический контекст поведения моделируемой системы. Поэтому при построении диаграмм состояний необходимо использовать специальные понятия, которые и будут рассмотрены в данной главе.

Ранее было отмечено, что каждая прикладная система характеризуется не только структурой составляющих ее элементов, но и некоторым поведением или функциональностью. Для общего представления функциональности моделируемой системы предназначены диаграммы вариантов использования, которые на концептуальном уровне описывают поведение системы в целом.

Сейчас наша задача заключается в том, чтобы представить наиболее общее поведение на логическом уровне, тем самым раскрыть сущность ответа на вопрос: "В процессе какого поведения система реализует необходимую пользователям функциональность?"

В общем случае для моделирования поведения на логическом уровне в языке UML могут использоваться сразу несколько канонических диаграмм: состояний, деятельности, последовательности и кооперации, каждая из которых фиксирует внимание на отдельном аспекте функционирования системы. В отличие от других диаграмм диаграмма состояний описывает процесс изменения состояний системы или ее подсистемы при реализации всех вариантов использования. При этом изменение состояний отдельных элементов системы может быть вызвано внешними воздействиями со стороны других элементов или извне системы. Именно для описания реакции системы на подобные внешние воздействия и используются диаграммы состояний.

Главное предназначение этой диаграммы — описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение моделируемой системы в течение всего ее жизненного цикла. Диаграмма состояний представляет динамическое поведение сущностей, на основе спецификации их реакции на восприятие некоторых конкретных событий. Системы, которые реагируют на внешние действия от других систем или от пользователей, иногда называют *реактивными*. Если такие действия инициируются в произвольные случайные моменты, то говорят об *асинхронном* поведении модели.

Хотя диаграммы состояний чаще всего используются для описания поведения отдельных систем и подсистем, они также могут быть применены для спецификации функциональности экземпляров отдельных классов (объектов), т. е. для моделирования всех возможных изменений состояний конкретных объектов.

Диаграмма состояний по существу является графом специального вида, который служит для представления некоторого конечного автомата. Хотя понятие конечного автомата основывается на общей теории автоматов, в контексте языка UML оно обладает дополнительной семантикой. Вершинами графа конечного автомата в контексте языка UML являются состояния и некоторые другие типы элементов модели (псевдосостояния), которые изображаются соответствующими графическими символами. Дуги графа служат для обозначения переходов из состояния в состояние. Диаграммы состояний могут быть вложены друг в друга, образуя вложенные диаграммы для более детального представления состояний отдельных элементов модели. Для понимания семантики конкретной диаграммы состояний необходимо представлять не только особенности поведения моделируемой сущности, но и знать общие сведения из теории конечных автоматов.

8.1. Конечные автоматы

Конечный автомат (state machine) в языке UML представляет собой некоторый формализм для моделирования поведения отдельных элементов модели или системы в целом. В метамодели UML конечный автомат является пакетом, в котором определено множество понятий, необходимых для представления поведения моделируемой сущности в виде дискретного пространства с конечным числом состояний и переходов. С другой стороны, конечный автомат описывает поведение отдельного объекта в форме последовательности состояний, которые охватывают все этапы его жизненного цикла, начиная от создания объекта и заканчивая его уничтожением. Каждая диаграмма состояний представляет собой некоторый конечный автомат.

Простейшим примером визуального представления состояний и переходов на основе формализма конечных автоматов может служить рассмотренная выше ситуация с исправностью некоторого технического устройства, например, персонального компьютера (ПК). В общем случае можно рассмотреть два его состояния: "исправен" и "неисправен" и два перехода: "выход из строя" и "ремонт". Графически эта информация может быть представлена в виде изображенной ниже диаграммы состояний ПК (рис. 8.1).

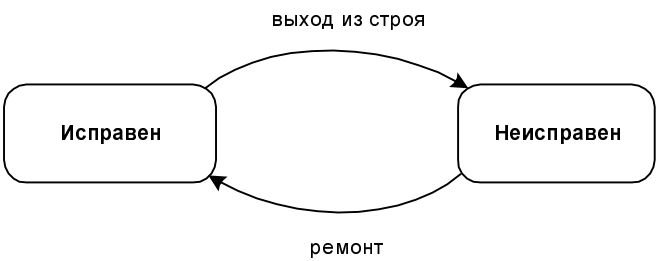


Рис. 8.1. Простейший пример диаграммы состояний для технического устройства типа персональный компьютер

Основными понятиями, входящими в формализм конечного автомата, являются *состояние* и *переход*. Основное различие между ними заключается в том, что длительность нахождения системы в отдельном состоянии существенно превышает время, которое затрачивается на переход из одного состояния в другое. Предполагается, что в пределе время перехода из одного состояния в другое равно нулю (если дополнительно ничего не сказано). Другими словами, переход объекта из состояния в состояние происходит мгновенно.

В общем случае конечный автомат представляет динамические аспекты моделируемой системы в виде ориентированного графа, вершины которого соответствуют состояниям, а дуги — переходам. При этом поведение моде-

лируется как последовательное перемещение по графу состояний от вершины к вершине, по связывающим их дугам с учетом их ориентации. Для графа состояний системы можно ввести в рассмотрение специальные свойства.

Одним из таких свойств является выделение из всей совокупности двух специальных состояний: начального и конечного. Хотя ни в графе состояний, ни на диаграмме состояний время нахождения системы в том или ином состоянии явно не учитывается, предполагается, что последовательность изменения состояний упорядочена во времени. Другими словами, каждое последующее состояние наступает позже предшествующего ему состояния.

Еще одним свойством графа состояний может служить *достижимость* состояний. Речь идет о том, что навигация или ориентированный путь в графе состояний определяет специальное бинарное отношение на множестве всех состояний системы. Это отношение характеризует потенциальную возможность перехода системы из рассматриваемого состояния в некоторое другое состояние. Очевидно, для достижимости состояний необходимо наличие связывающего их ориентированного пути в графе состояний.

Формализм конечных автоматов допускает вложение одних в другие для уточнения внутренней структуры отдельных более общих состояний (макросостояний). В этом случае вложенные конечные автоматы получили название *конечных подавтоматов*. Подавтоматы могут использоваться для внутренней спецификации процедур и функций, реализация которых обуславливает поведение моделируемой системы или объекта. Например, состояние неисправности технического устройства (см. рис. 8.1) может быть детализировано на отдельные подсостояния, каждое из которых может характеризовать неисправность отдельных подсистем, входящих в состав данного устройства.

В языке UML понятие конечного автомата дополнено специальной семантикой входящих в соответствующий пакет элементов. Далее в этой главе будут рассмотрены основные элементы поведения, которые образуют концептуальный базис, необходимый для правильного построения диаграмм состояний.

Формализм обычного конечного автомата основан на выполнении следующих обязательных условий.

1. Конечный автомат не запоминает историю перемещения из состояния в состояние. С точки зрения моделируемого поведения определяющим является сам факт нахождения моделируемого элемента в том или ином состоянии, но никак не последовательность состояний, в результате которой элемент перешел в текущее состояние. Другими словами, конечный автомат "забывает" все состояния, которые предшествовали текущему в данный момент. Образно говоря, существует непрозрачная стена, отделяющая текущее состояние от прошлой истории поведения объекта.

Примечание

Данное условие может быть изменено явным образом для сохранения некоторых аспектов предыстории поведения объекта на основе введения так называемых исторических состояний, которые будут описаны ниже в этой главе.

2. В каждый момент конечный автомат может находиться только в одном из своих состояний. Это означает, что формализм конечного автомата предназначен для моделирования последовательного поведения, когда моделируемый элемент в течение своего жизненного цикла последовательно проходит через все свои состояния. При этом конечный автомат может находиться в отдельном состоянии как угодно долго, если не происходит никаких событий.

Примечание

Это условие ограничивает область применения конечных автоматов только моделированием последовательных процессов. Необходимость моделирования параллельных процессов приводит к рассмотрению в контексте одной модели нескольких конечных подавтоматов, каждый из которых специфицирует отдельный подпроцесс поведения.

3. Хотя процесс изменения состояний конечного автомата происходит во времени, явно концепция времени не входит в формализм конечного автомата. Это означает, что длительность нахождения конечного автомата в том или ином состоянии, а также время достижения того или иного состояния никак не специфицируются. Другими словами, время на диаграмме состояний присутствует в неявном виде, хотя для отдельных событий может быть указан интервал времени и в явном виде.

Примечание

Концепция времени в явной форме учитывается при построении диаграммы последовательности, когда требуется синхронизировать во времени процессы взаимодействия нескольких объектов модели. Поскольку диаграмма состояний предназначена для моделирования поведения, которое определяется асинхронными событиями, эти события могут происходить в заранее неизвестные моменты.

4. Количество состояний конечного автомата должно быть обязательно конечным, и все они должны быть специфицированы явным образом. Это условие представляется тривиальным, поскольку в языке UML рассматриваются только конечные автоматы, однако отдельные псевдосостояния могут не иметь спецификаций (начальное и конечное состояния). В этом случае их назначение и семантика полностью определяются из контекста модели и рассматриваемой диаграммы состояний.

~~2.~~5. Граф конечного автомата не должен содержать изолированных состояний и переходов. Это условие означает, что для каждого из состояний,

кроме начального, должно быть определено хотя бы одно предшествующее состояние. Каждый переход должен обязательно соединять два состояния конечного автомата. Допускается переход из состояния в себя, такой переход еще называют рефлексивным переходом или "петлей".

Примечание

Что касается наличия у конечного автомата *тупиковых* состояний, т. е. таких состояний, за исключением конечного, для которых не определено ни одного последующего состояния, то потенциально они могут присутствовать на диаграмме, однако лучше их визуализировать явно с помощью конечных состояний.

~~3.~~6. Конечный автомат не должен содержать конфликтующих переходов, т. е. таких переходов из одного и того же состояния, когда при наступлении одного и того же события моделируемый элемент одновременно может перейти в два и более последующих состояния (кроме случая параллельных конечных подавтоматов). В языке UML исключение конфликтов возможно на основе введения так называемых сторожевых условий, которые еще будут рассмотрены.

Таким образом, правила поведения системы или отдельного объекта, моделируемого некоторым конечным автоматом, определяются, с одной стороны, общим формализмом конечного автомата, а с другой — его графическим изображением в форме конкретной диаграммы состояний. Хотя на диаграмме состояний не присутствует явно формализм конечных автоматов, разработчику следует помнить об указанных выше условиях 1—6, чтобы избежать досадных недоразумений при изображении диаграмм состояний. Как уже упоминалось ранее, основными графическими примитивами диаграммы состояний являются собственно состояния и переходы, семантические особенности которых рассматриваются далее в этой главе.

8.2. Состояние

Понятие состояния (state) является фундаментальным не только в метамодели языка UML, но и в прикладном системном анализе. Ранее, в *главе 1*, кратко были рассмотрены особенности представления динамических характеристик сложных систем, традиционно используемых для моделирования поведения. Вся концепция динамической системы основывается на понятии состояния системы. Однако семантика состояния в языке UML имеет целый ряд специфических особенностей.

В языке UML под состоянием понимается абстрактный метакласс, используемый для моделирования отдельной ситуации, в течение которой имеет место выполнение некоторого условия. Состояние может быть задано в виде набора конкретных значений атрибутов объекта некоторого класса, при этом изменение отдельных значений этих атрибутов будет отражать изменение состояния моделируемого объекта или системы в целом.

Следует заметить, что не каждый атрибут класса может характеризовать состояние его объектов. Как правило, имеют значение только такие свойства элементов системы, которые отражают динамический или функциональный аспект ее поведения. В этом случае состояние будет характеризоваться некоторым инвариантным условием, включающим в себя только значимые для поведения объекта или системы атрибуты классов и их значения.

Например, такое условие может представлять статическую ситуацию, когда объект находится в состоянии ожидания возникновения некоторого внешнего события. С другой стороны, аналогичное условие используется для моделирования динамических аспектов, когда в ходе нахождения объекта в некотором состоянии выполняются некоторые действия. В последнем случае соответствующая деятельность начинается в момент перехода моделируемого элемента в рассматриваемое состояние, и элемент может покинуть данное состояние в момент завершения этой деятельности.

Состояние на диаграмме изображается прямоугольником с закругленными вершинами (рис. 8.2). Этот прямоугольник, в свою очередь, может быть разделен на две секции горизонтальной линией. Если указана лишь одна секция, то в ней записывается только имя состояния (рис. 8.2, *а*). В противном случае в первой из них записывается имя состояния, а во второй — список некоторых внутренних действий или переходов в данном состоянии (рис. 8.2, *б*). При этом под действием в языке UML понимают некоторую атомарную операцию, выполнение которой приводит к изменению состояния или возврату некоторого значения (например, "истина" или "ложь").

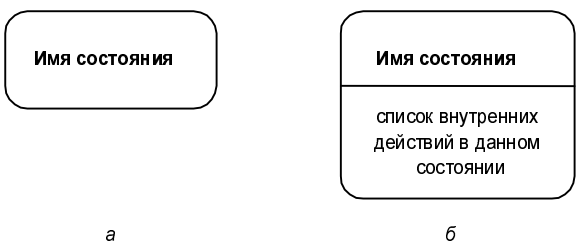


Рис. 8.2. Графическое изображение состояний на диаграмме состояний

8.2.1. Имя состояния

Имя состояния представляет собой строку текста, которая раскрывает содержательный смысл или семантику данного состояния. Имя должно представлять собой законченное предложение и всегда записываться с заглавной буквы. Поскольку состояние системы является составной частью процесса ее функционирования, рекомендуется в качестве имени использовать глаголы в настоящем времени (звенит, печатает, ожидает) или соответствующие причастия (занят, свободен, передано, получено). Как исключение, имя

у состояния может отсутствовать, т. е. оно является необязательным для некоторых состояний. В этом случае состояние является анонимным и если на диаграмме состояний их несколько, то все они должны различаться между собой.

8.2.2. Список внутренних действий

Для некоторых состояний может потребоваться дополнительно указать некоторые действия, которые должны быть выполнены моделируемым элементом при нахождении его в том или ином состоянии. Для этой цели служит дополнительная секция в обозначении состояния, содержащая перечень внутренних действий или деятельность, которые выполняются в процессе нахождения моделируемого элемента в данном состоянии. Каждое из действий записывается в виде отдельной строки и имеет следующий формат:

<метка действия ‘/’ выражение действия>

Метка действия указывает на обстоятельства или условия, при которых будет выполняться деятельность, определенная выражением действия. При этом выражение действия может использовать любые атрибуты и связи, которые принадлежат области имен или контексту моделируемого объекта. Если список выражений действия пустой, то метка действия с разделителем в виде наклонной черты " ‘/’ " не указываются.

Перечень меток действий в языке UML фиксирован (причем эти метки не могут быть использованы в качестве имен событий):

- entry — указывает на то, что следующее за ней выражение действия должно быть выполнено в момент входа в данное состояние (входное действие);
- exit — указывает на то, что следующее за ней выражение действия должно быть выполнено в момент выхода из данного состояния (выходное действие);
- do — специфицирует некоторую деятельность (do activity) или так называемую *ду-деятельность*, которая выполняется в течение всего времени, пока объект находится в данном состоянии, или до тех пор, пока не будет выполнено условие ее окончания, специфицированное в соответствующей операции класса или вычислительной процедуре. В последнем случае при завершении деятельности генерируется соответствующее событие;
- include — используется для обращения к конечному подавтомату, при этом следующее за ней выражение действия содержит имя этого подавтомата.

Во всех остальных случаях метка действия идентифицирует событие, которое запускает соответствующее выражение действия. Эти события называются *внутренними переходами*. Семантически они эквивалентны рефлексивным переходам для данного состояния, за исключением той особенности,

что и выход из этого состояния или повторный вход в него не происходит. Это означает, что действия входа и выхода не выполняются.

Примечание

Принято считать, что выполнение внутренних действий в состоянии не может быть прервано никакими внешними событиями, в отличие от внутренней деятельности, выполнение которой требует определенного времени.

В качестве примера состояния рассмотрим ситуацию ввода пароля пользователем при аутентификации входа в некоторую программную систему (рис. 8.3). В этом случае список внутренних действий в данном состоянии не пуст и включает 3 отдельных действия. Первое и третье действия стандартные и описаны ранее, а действие с меткой `символ` обеспечивает выполнение операции получения символа с клавиатуры.

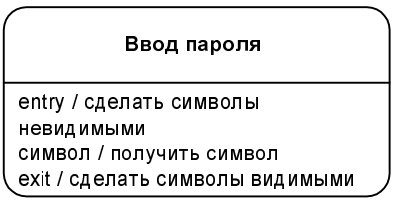
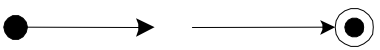


Рис. 8.3. Пример состояния с непустой секцией внутренних действий

8.2.3. Начальное состояние

Начальное состояние (initial state) представляет собой частный случай состояния, которое не содержит никаких внутренних действий и поэтому относится к категории псевдосостояния. В этом состоянии находится объект по умолчанию в начальный момент. Оно служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний. Графически начальное состояние в языке UML обозначается в виде закрашенного кружка (рис. 8.4, *а*), из которого может только выходить стрелка-переход.



а *б*

начальное состояние конечное состояние

Рис. 8.4. Графическое изображение начального и конечного состояний на диаграмме состояний

На самом верхнем уровне представления объекта переход из начального состояния может быть помечен событием создания (инициализации) данного объекта. В противном случае этот переход никак не помечается. Если этот переход не помечен, то он является первым переходом на диаграмме состояний в следующее за ним состояние. Каждая диаграмма или поддиаграмма состояний должна иметь единственное начальное состояние.

8.2.4. Конечное состояние

Конечное (финальное) состояние (final state) представляет собой частный случай состояния, которое также не содержит никаких внутренних действий и поэтому также является псевдосостоянием. В этом состоянии должен находиться моделируемый объект или система по умолчанию после завершения работы конечного автомата. Оно служит для указания на диаграмме состояний графической области, в которой завершается процесс изменения состояний или жизненный цикл данного объекта. Графически конечное состояние в языке UML обозначается в виде закрашенного кружка, помещенного в окружность (рис. 8.4, *б*), в которую может только входить стрелка-переход. Каждая диаграмма состояний или подсостояний может иметь несколько конечных состояний, при этом все они считаются эквивалентными на одном уровне вложенности состояний.

8.3. Переход

Простой переход (simple transition) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния другим. Пребывание моделируемого объекта или системы в первом состоянии может сопровождаться выполнением некоторых внутренних действий (деятельности), при этом переход в другое состояние будет возможен либо после завершения этих действий (деятельности), либо при возникновении некоторых событий. В обоих случаях говорят, что переход срабатывает, или происходит срабатывание перехода. До срабатывания перехода объект находится в предыдущем от него состоянии, называемом исходным состоянием, или в источнике (не путать с начальным состоянием — это разные понятия), а после срабатывания — в последующем от него состоянии (целевом состоянии).

Переход осуществляется при наступлении некоторого события: окончания выполнения деятельности (do activity), получении объектом сообщения или приемом сигнала. На переходе указывается имя события. Кроме того, на переходе могут указываться действия, производимые объектом в ответ на внешние события при переходе из одного состояния в другое. Срабатывание перехода может зависеть не только от наступления некоторого события, но и от выполнения определенного условия, называемого сторожевым условием.

Объект перейдет из одного состояния в другое в том случае, если произошло указанное событие и сторожевое условие приняло значение "истина".

Примечание

Переход может быть направлен в то же состояние, из которого он выходит. В этом случае его называют *переходом в себя*. Исходное и целевое состояния перехода в себя совпадают. Этот переход изображается петлей со стрелкой и отличается от внутреннего перехода. При переходе в себя объект покидает исходное состояние, а затем снова входит в него. При этом всякий раз выполняются внутренние действия, специфицированные метками entry и exit.

На диаграмме состояний переход изображается сплошной линией со стрелкой, которая выходит из исходного состояния и направлена в целевое состояние (например, `выход из строя` на рис. 8.1). Каждый переход может быть помечен строкой текста, которая имеет следующий общий формат:

<*сигнатура события*>’[‘<*сторожевое условие*>’]’ <*выражение действия*>.

При этом *сигнатура события* описывает некоторое событие с необходимыми аргументами:

<*имя события*>’(‘<*список параметров, разделенных запятыми*>’)’.

8.3.1. Событие

Термин событие (event) требует отдельного пояснения, поскольку является самостоятельным элементом языка UML. Формально, событие представляет собой спецификацию некоторого факта, имеющего место в пространстве и во времени. Про события говорят, что они "происходят", при этом отдельные события должны быть упорядочены во времени. После наступления некоторого события нельзя уже вернуться к предыдущим событиям, если такая возможность не предусмотрена явно в модели.

Семантика понятия события фиксирует внимание на внешних проявлениях качественных изменений, происходящих при переходе моделируемого объекта из состояния в состояние. Например, при включении электрического переключателя происходит некоторое событие, в результате которого комната становится освещенной. После успешного ремонта компьютера также происходит немаловажное событие — восстановление его работоспособности. Если поднять трубку обычного телефона, то, в случае его исправности, мы ожидаем услышать тоновый сигнал. И этот факт тоже является событием.

В языке UML события играют роль стимулов, которые инициируют переходы из одних состояний в другие. В качестве событий можно рассматривать сигналы, вызовы, окончание фиксированных промежутков времени или моменты окончания выполнения определенных действий. В зависимости

от вида происходящих событий-стимулов в языке UML различают два типа переходов: триггерные и нетриггерные.

- Переход называется *триггерным*, если его специфицирует некоторое событие-триггер. В этом случае рядом со стрелкой триггерного перехода обязательно указывается имя события в форме строки текста, начинающейся со строчной буквы. Наиболее часто в качестве имен триггерных переходов задают имена операций, вызываемых у тех или иных объектов системы. После имени такого события следуют круглые скобки для явного задания параметров соответствующей операции. Если таких параметров нет, то список со скобками может отсутствовать. Например, переходы на рис. 8.1 являются триггерными, поскольку с каждым из них связано некоторое событие-триггер, происходящее асинхронно в момент выхода из строя технического устройства или в момент окончания его ремонта.
- Переход называется *нетриггерным*, если он происходит по завершении выполнения действий (деятельности) в исходном состоянии. Нетриггерные переходы часто называют переходами по завершении ду-деятельности. Для них рядом со стрелкой перехода не указывается никакого имени события, а в исходном состоянии должна быть описана внутренняя ду-деятельность, по завершении которой произойдет тот или иной нетриггерный переход.

8.3.2. Сторожевое условие

Сторожевое условие (guard condition), если оно есть, всегда записывается в прямых скобках и представляет собой некоторое булевское выражение. Напомним, что булевское выражение должно принимать одно их двух значений — "истина" или "ложь". Из контекста диаграммы состояний должна явно следовать семантика этого выражения, а для записи выражения может использоваться обычный язык, псевдокод или язык программирования.

Дополнение триггерных и нетриггерных переходов сторожевыми условиями позволяет явно специфицировать семантику их срабатывания. Если сторожевое условие принимает значение "истина", то соответствующий переход при наступлении события-триггера или завершения деятельности может сработать, в результате чего объект перейдет в целевое состояние. Если сторожевое условие принимает значение "ложь", то переход не может сработать, даже если произошло событие-триггер или завершилась деятельность в исходном переходе. Очевидно, в случае невыполнения сторожевого условия моделируемый объект или система останется в исходном состоянии. Однако вычисление истинности сторожевого условия в модели происходит только после возникновения ассоциированного с ним события-триггера или завершения деятельности, которые инициируют соответствующий переход.

Поскольку общее количество выходящих из состояния переходов в языке UML никак не ограничено (но конечно), то не исключена ситуация, когда из одного состояния могут выходить несколько переходов с одним и тем же событием-триггером. Каждый из таких переходов должен содержать собственное сторожевое условие, но при этом никакие два или более сторожевых условий не должны одновременно принимать значение "истина". В противном случае на диаграмме состояний будет иметь место *конфликт триггерных переходов*, что делает несостоятельной (ill formed) модель системы в целом.

Примечание

Аналогичное замечание справедливо и для нетриггерных переходов, когда из одного состояния выходят несколько переходов по завершению деятельности. Каждый из таких переходов также должен содержать собственное сторожевое условие, при этом никакие два или более сторожевых условий не должны одновременно принимать значение "истина". В противном случае на диаграмме состояний будет иметь место *конфликт нетриггерных переходов*, что также делает несостоятельной (ill formed) модель системы в целом.

Примером события-триггера может служить разрыв телефонного соединения с провайдером Интернет-услуг после окончания загрузки электронной почты клиентской почтовой программой при удаленном доступе к Интернету. В этом случае сторожевое условие есть не что иное, как ответ на вопрос "Пуст ли почтовый ящик клиента на сервере провайдера?" В случае ответа "истина", следует отключить соединение с провайдером, что и делает автоматически почтовая программа-клиент. В случае отрицательного ответа на данный вопрос сторожевое условие принимает значение "ложь", тем самым указывая о необходимости оставаться в состоянии загрузки почты и не разрывать телефонное соединение.

Графически логика моделирования почтовой программы может быть представлена в виде следующего фрагмента диаграммы состояний (рис. 8.5). Как можно заключить из контекста, в начальном состоянии программа не выполняется, хотя и имеется на компьютере пользователя. В момент ее запуска происходит активизация. В этом состоянии программа может находиться неопределенно долго, пока пользователь ее не закроет, т. е. не выгрузит из оперативной памяти компьютера. После окончания активизации программа переходит в конечное состояние. В активном состоянии программы пользователь может читать сообщения электронной почты, создавать собственные послания и выполнять другие действия, явно не указанные на диаграмме.

Однако при необходимости получить новую почту, пользователь должен установить телефонное соединение с провайдером, что и показано явно на диаграмме верхним переходом. Другими словами, пользователь инициирует событие-триггер `установить телефонное соединение()`. В качестве параметра этого события выступает конкретный `телефонный номер` модемного пула провайдера. Далее следует проверка сторожевого условия `телефонное соединение установлено`, которое следует понимать как вопрос. Только

в случае положительного ответа "да" (т. е. "истина"), происходит переход почтовой программы-клиента из состояния `Активизация почтовой программы` в состояние `Загрузка почты с сервера провайдера`. В противном случае (линия занята, неверный ввод пароля, отключенный логин) никакой загрузки почты не произойдет, и программа останется в прежнем своем состоянии.

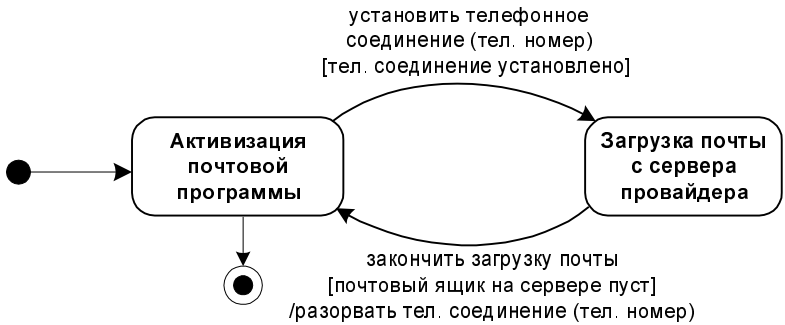


Рис. 8.5. Фрагмент диаграммы состояний для моделирования почтовой программы-клиента

Второй триггерный переход на диаграмме инициирует автоматический разрыв телефонного соединения с провайдером после окончания загрузки почты на компьютер пользователя. В этом случае событие-триггер `закончить загрузку почты` происходит после проверки сторожевого условия `почтовый ящик на сервере пуст`, которое также следует понимать в форме вопроса. При положительном ответе на этот вопрос (результат — "истина"), когда вся почта загружена или ее просто нет в ящике, почтовая программа прекращает загрузку почты и возвращается в прежнее состояние активизации. В случае же отрицательного ответа загрузка почты будет продолжена.

Примечание

Конечно, рассмотренный пример имеет методический характер и иллюстрирует лишь основные особенности поведения почтовой программы-клиента в одном из ее аспектов. И даже этот аспект загрузки почты во многом условный, поскольку не учитывает реакцию программы на такие сообщения, как "линия занята" или самопроизвольный разрыв соединения.

В отдельных случаях может произойти редкое, но весьма неприятное событие, получившее название "залипание модема". Это характерно для ситуации, когда вся почта загружена, а конечный автоматический разрыв соединения не происходит. Тем не менее и этот случай можно предусмотреть в нашей модели, дополнив диаграмму еще одним переходом с аналогичным событием-триггером `закончить загрузку почты` и с новым сторожевым условием. Это сторожевое условие должно проверять максимально допустимое

время соединения для загрузки почты (например, 600 секунд) и может быть сформулировано следующим образом — `время загрузки почты превышает 600 секунд`. Модифицировать диаграмму состояний для этого случая читателям предлагается самостоятельно в качестве упражнения.

8.3.3. Выражение действия

Выражение действия (action expression) выполняется только в том случае, когда переход срабатывает. Представляет собой вызов операции или передачу некоторого сообщения, имеет атомарный характер и выполняется сразу после срабатывания соответствующего перехода до начала каких бы то ни было действий в целевом состоянии. Атомарность действия означает, что оно не может быть прервано никаким другим действием до тех пор, пока не закончится его выполнение. Данное действие может оказывать влияние как на сам объект, так и на его окружение, если это с очевидностью следует из контекста модели. Данное выражение записывается после знака "/" в строке текста, присоединенной к соответствующему переходу.

В общем случае, выражение действия может содержать целый список отдельных действий, разделенных символом ";". Обязательное требование – все действия из списка должны четко различаться между собой и следовать в порядке их записи. На синтаксис записи выражений действия не накладывается никаких ограничений. Главное — их запись должна быть понятна разработчикам модели и программистам. Поэтому чаще всего выражения записывают на одном из языков программирования, который предполагается использовать для реализации модели.

В качестве примера выражения действия (рис. 8.5) может служить действие `разорвать телефонное соединение(телефонный номер)`, которое должно быть выполнено сразу после выполнения перехода `закончить загрузку почты` при истинности сторожевого условия `почтовый ящик на сервере пуст`.

Другим примером может служить очевидная ситуация с выделением графических объектов на экране монитора при однократном нажатии левой кнопки мыши. Имеется в виду обработка сигналов от пользователя при выделении тех или иных графических примитивов (пиктограмм). В этом случае соответствующий переход может иметь следующую строку текста:

```
нажата и отпущена левая кнопка мыши(координаты)[координаты
в области графического объекта]/ выделить объект(цвет)
```

Результатом этого триггерного перехода может быть, например, активизация некоторых свойств объекта (указание размера выделенного файла в строке состояния или последующее его удаление в корзину).

Примечание

Иногда после выражения действия может быть записано сообщение-сигнал в формате: '^' <*имя объекта приемника сообщения*> '.' <*имя посылаемого*

сообщения> ’(‘<*параметр*>’:’<*тип*>,’)’. При этом сообщение имеет чисто информационный характер и не передает управление на объект-приемник сообщения.

8.4. Составное состояние и подсостояние

Составное состояние (composite state) — такое сложное состояние, которое состоит из других вложенных в него состояний. Последние выступают по отношению к первому как *подсостояния* (substate). Хотя между ними имеет место отношение композиции, графически все вершины диаграммы, которые соответствуют вложенным состояниям, изображаются внутри символа составного состояния (рис. 8.6). В этом случае размеры графического символа составного состояния увеличиваются так, чтобы вместить в себя все подсостояния.

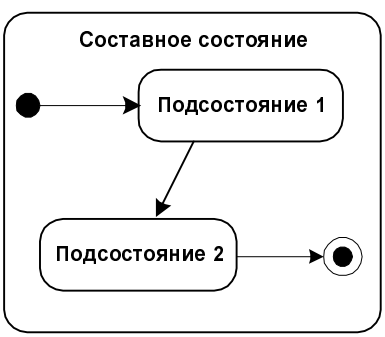


Рис. 8.6. Графическое представление составного состояния с двумя вложенными в него последовательными подсостояниями

Составное состояние, которое называют также *суперсостоянием* или *состоянием-композитом*, может содержать или несколько последовательных подсостояний, или несколько параллельных конечных подавтоматов. Каждое суперсостояние может уточняться только одним из указанных способов. При этом любое из подсостояний, в свою очередь, может являться составным состоянием и содержать внутри себя другие вложенные подсостояния. Количество уровней вложенности составных состояний в языке UML не фиксировано.

8.4.1. Последовательные подсостояния

Последовательные подсостояния (sequential substates) используются для моделирования такого поведения объекта, во время которого в каждый момент объект может находиться в одном и только одном подсостоянии. Поведение объекта в этом случае представляет собой последовательную смену подсостояний,

начиная от начального и заканчивая конечным подсостоянием. Хотя моделируемый объект или система продолжает находиться в составном состоянии, введение в рассмотрение последовательных подсостояний позволяет учесть более тонкие логические аспекты его внутреннего поведения.

Как пример рассмотрим в качестве моделируемой системы обычный телефонный аппарат. Он может находиться в различных состояниях, одним из которых является состояние дозвона до абонента. Очевидно, для того чтобы позвонить, необходимо снять телефонную трубку, услышать тоновый сигнал, после чего набрать нужный телефонный номер. Таким образом, состояние дозвона до абонента является составным и состоит из двух последовательных подсостояний: `Поднять телефонную трубку` и `Набрать телефонный номер`. Фрагмент диаграммы состояний для этого примера содержит одно составное состояние и два последовательных подсостояния (рис. 8.7).

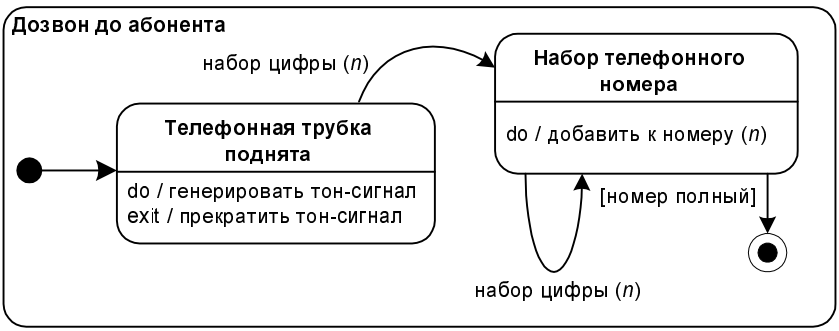


Рис. 8.7. Пример составного состояния с двумя вложенными последовательными подсостояниями

Некоторых пояснений могут потребовать переходы. Два из них специфицируют событие-триггер, которое имеет имя: `набор цифры(n)` с параметром *n*. В качестве параметра, как нетрудно предположить, выступает отдельная цифра на диске телефонного аппарата. Переход из начального подсостояния нетриггерный, поскольку он не содержит никакой строки текста. Последний переход в конечное подсостояние также не имеет события-триггера, но имеет сторожевое условие, проверяющее правильность набранного номера абонента. Только в случае истинности этого условия телефонный аппарат может перейти в конечное подсостояние для суперсостояния `Дозвон до абонента`.

Каждое составное состояние должно содержать в качестве вложенных подсостояний начальное и конечное. При этом начальное подсостояние является исходным, когда происходит переход объекта в данное составное состояние. Если составное состояние содержит внутри себя конечное (финальное)

подсостояние, то переход в это вложенное конечное состояние означает завершение нахождения объекта в данном суперсостоянии. Важно помнить, что для последовательных подсостояний начальное и конечное состояния должны быть единственными в каждом составном состоянии.

Это можно объяснить следующим образом. Каждая совокупность вложенных последовательных подсостояний представляет собой конечный подавтомат того конечного автомата, которому принадлежит рассматриваемое составное состояние. Поскольку каждый конечный автомат может иметь по определению единственное начальное и единственное конечное состояния, то для любого его конечного подавтомата это условие также должно выполняться (рис. 8.7).

8.4.2. Параллельные подсостояния

Параллельные подсостояния (concurrent substates) позволяют специфицировать два и более конечных подавтомата, которые могут выполняться параллельно внутри составного события. Каждый из конечных подавтоматов занимает некоторую область (регион) внутри составного состояния, которая отделяется от остальных горизонтальной пунктирной линией. Если на диаграмме состояний имеется составное состояние с вложенными параллельными подсостояниями, то объект может одновременно находиться в каждом из этих подсостояний.

Отдельные параллельные подсостояния могут, в свою очередь, состоять из нескольких последовательных подсостояний (конечные подавтоматы 1 и 2 на рис. 8.8). В этом случае по определению объект может находиться только в одном из последовательных подсостояний конечного подавтомата. Таким образом, для абстрактного примера (рис. 8.8) допустимо одновременное нахождение объекта в подсостояниях (1, 3, 4), (2, 3, 4), (1, 3, 5), (2, 3, 5). Недопустимо нахождение объекта одновременно в подсостояниях (1, 2 ,3) или (3, 4, 5).

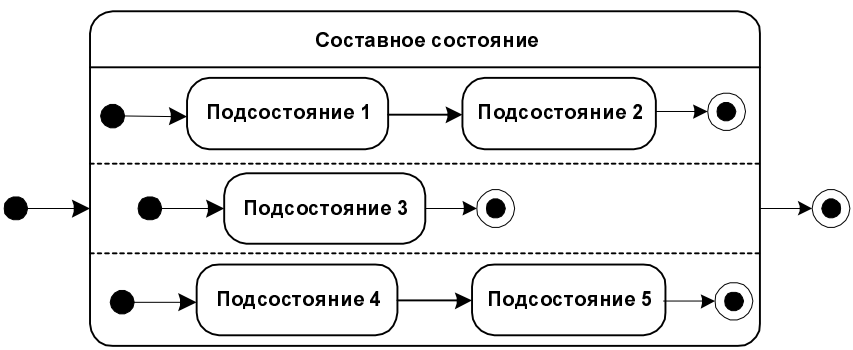


Рис. 8.8. Графическое изображение составного состояния с вложенными параллельными подсостояниями

Поскольку каждый регион вложенного состояния специфицирует некоторый конечный подавтомат, то для каждого из вложенных конечных подавтоматов могут быть определены собственные начальное и конечное состояния (рис. 8.8). При переходе в данное составное состояние каждый из конечных подавтоматов оказывается в своем начальном состоянии. Далее происходит параллельное выполнение каждого из этих конечных подавтоматов, причем выход из составного состояния будет возможен лишь в том случае, когда все конечные подавтоматы будут находиться в своих конечных состояниях.

Если какой-либо из конечных подавтоматов пришел в свое финальное состояние раньше других, то он должен ожидать, пока и другие не придут в свои финальные состояния.

В некоторых случаях бывает желательно скрыть внутреннюю структуру составного состояния. Например, отдельный конечный подавтомат, специфицирующий составное состояние, может быть настолько большим по масштабу, что его визуализация затруднит общее представление диаграммы состояний. В подобной ситуации допускается не раскрывать на исходной диаграмме состояний данное составное состояние, а указать в правом нижнем углу специальный символ-пиктограмму (символ составного состояния) (рис. 8.9). В последующем диаграмма состояний для соответствующего конечного подавтомата может быть изображена отдельно от основной с необходимыми комментариями.

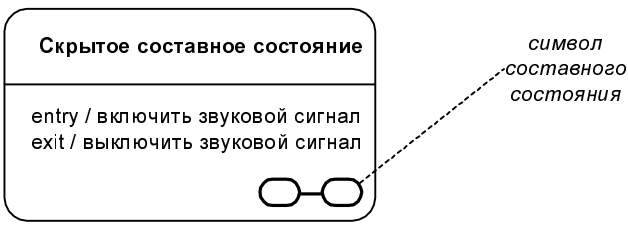


Рис. 8.9. Составное состояние со скрытой внутренней структурой и специальной пиктограммой

8.5. Исторические состояния

Как было отмечено ранее, формализм обычного конечного автомата не позволяет учитывать предысторию в процессе моделирования поведения систем и объектов. Однако функционирование целого ряда систем основано на возможности выхода из отдельного суперсостояния с последующим возвращением в это же состояние. При этом может оказаться необходимым учесть ту часть деятельности, которая была выполнена на момент выхода из этого состояния, чтобы не начинать ее выполнение сначала. Для этой цели в языке UML существует историческое состояние.

Историческое состояние (history state) применяется только в контексте составного состояния. Оно используется для запоминания того из последовательных подсостояний, которое было текущим в момент выхода из составного состояния. При этом существует две разновидности исторического состояния: неглубокое (недавнее) и глубокое (давнее) (рис. 8.10).

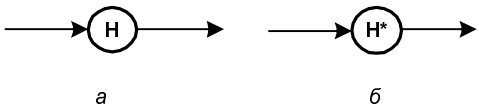


Рис. 8.10. Графическое изображение недавнего (*а*) и давнего (*б*) исторического состояния

Неглубокое историческое состояние (shallow history state) обозначается в форме небольшой окружности, в которую помещена латинская буква "H" (рис. 8.10, *а*). Это состояние обладает следующей семантикой. Во-первых, оно является первым подсостоянием в составном состоянии, и переход извне в рассматриваемое составное состояние должен вести непосредственно в данное историческое состояние. Во-вторых, при первом попадании в неглубокое историческое состояние оно не хранит никакой истории (история пуста). Другими словами, при первом переходе в недавнее историческое состояние оно заменяет собой начальное состояние соответствующего конечного подавтомата.

Далее может следовать последовательное изменение вложенных подсостояний. Если в некоторый момент происходит выход из составного состояния (например, в случае наступления некоторого события), то рассматриваемое историческое состояние запоминает (сохраняет) то из подсостояний, которое являлось текущим на момент выхода из данного составного состояния. При последующем входе в это составное состояние неглубокое историческое подсостояние имеет непустую историю и сразу отправляет конечный подавтомат в сохраненное подсостояние, минуя все предшествующие ему подсостояния.

Историческое состояние теряет свою историю в тот момент, когда конечный подавтомат доходит до своего конечного состояния. При этом неглубокое историческое состояние запоминает историю только того конечного подавтомата, к которому оно относится. Другими словами, этот тип псевдосостояния способен запомнить историю только одного с ним уровня вложенности.

Если сохраненное подсостояние также является составным состоянием, а при выходе из исходного составного состояния необходимо запомнить некоторое подсостояние второго уровня вложенности, то в этом случае следует воспользоваться более сильным псевдосостоянием — глубоким историческим состоянием.

Глубокое историческое состояние (deep history state) также обозначается в форме небольшой окружности, в которую помещена латинская буква "H"

с дополнительным символом "*" (звездочка) (рис. 8.10, *б*), и служит для запоминания всех подсостояний любого уровня вложенности для исходного составного состояния.

Простым примером, иллюстрирующим применение неглубокого исторического состояния, может служить логика работы почтовой программы-клиента. Предположим, при запуске этой программы мы находимся в состоянии написания нового сообщения, причем набран уже значительный фрагмент текста. Почтовая программа может быть сконфигурирована таким образом, что в фиксированные моменты (например, каждые 30 минут) она проверяет наличие новых сообщений на сервере провайдера при удаленном доступе. Очевидно, что очередной дозвон, хотя и прерывает работу редактора (пользователя), не должен привести к потере набранного фрагмента текста.

В этом случае составное состояние `Работа редактора` должно содержать вложенное историческое состояние, которое запоминает выполненную работу. После окончания дозвона и загрузки новой почты (в случае ее наличия) мы должны вернуться к сохраненному фрагменту нашего текста и продолжить работу с редактором.

Рассмотренный ранее фрагмент диаграммы состояний почтовой программы-клиента (см. рис. 8.5) может быть дополнен с учетом указанного аспекта ее поведения. Читателю предлагается это проделать самостоятельно в качестве упражнения.

8.6. Сложные переходы

Рассмотренное выше понятие перехода является вполне достаточным для большинства типичных расчетно-вычислительных задач. Однако современные программные системы могут реализовывать очень сложную логику поведения отдельных своих компонентов. Может оказаться, что для адекватного представления процесса изменения состояний семантика обычного перехода для них недостаточна. С этой целью в языке UML специфицированы дополнительные обозначения и свойства, которыми могут обладать отдельные переходы на диаграмме состояний.

8.6.1. Переходы между параллельными состояниями

В отдельных случаях возникает необходимость явно показать ситуацию, когда некоторый переход может иметь несколько исходных состояний или несколько целевых состояний. Такой переход получил специальное название — *параллельный* переход. Введение в рассмотрение параллельных переходов может быть обусловлено необходимостью синхронизировать и/или разделить отдельные процессы управления на параллельные нити без спецификации дополнительной синхронизации в параллельных конечных подавтоматах.

Графически такой переход изображается вертикальной черточкой, аналогично обозначению перехода в известном формализме сетей Петри. Если параллельный переход имеет две или более входящие дуги (рис. 8.11, *а*), то его называют *слиянием* (join). Если же он имеет две или более исходящих из него дуг (рис. 8.11, *б*), то его называют *разделением* (fork). Текстовая строка спецификации параллельного перехода записывается рядом с черточкой и относится ко всем входящим (исходящим) дугам.

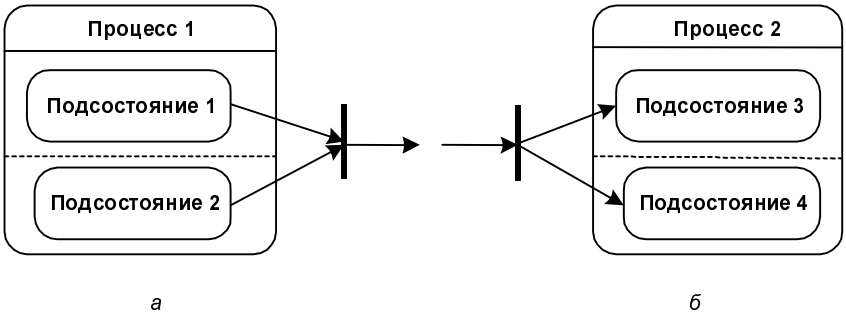


Рис. 8.11. Графическое изображение параллельного перехода из параллельных подсостояний (*а*) и параллельного перехода в параллельные подсостояния (*б*)

Срабатывание параллельного перехода происходит следующим образом. В первом случае переход-слияние срабатывает, если имееет место событие-триггер для всех исходных состояний этого перехода и выполнено (при его наличии) сторожевое условие. При срабатывании перехода-слияния одновременно покидаются все исходные подсостояния перехода (подсостояния 1 и 2) и происходит переход в целевое состояние. При этом каждое из исходных подсостояний перехода должно принадлежать отдельному конечному подавтомату, входящему в состав составного конечного автомата (процессу 1).

Во втором случае происходит разделение составного конечного автомата на два конечных подавтомата, образующих параллельные ветви вложенных подпроцессов. При этом после срабатывания перехода-разделения моделируемая система или объект одновременно будет находиться во всех целевых подсостояниях этого параллельного перехода (состояния 3 и 4). Далее процесс изменения состояний будет протекать согласно ранее рассмотренным правилам для составных состояний.

8.6.2. Переходы между составными состояниями

Переход, стрелка которого соединена с границей некоторого составного состояния, обозначает переход в это составное состояние (переход *b* на рис. 8.12). Он эквивалентен переходу в начальное состояние каждого из конечных подавтоматов (возможно, единственному), входящих в состав данного суперсостояния. Переход, выходящий из составного состояния (переходы *f* и *g*

на рис. 8.12), относится к каждому из вложенных подсостояний. Это означает, что система или объект может покинуть данное составное состояние, находясь в любом из его подсостояний. Для этого вполне достаточно наступления триггерного события и выполнения (в случае его наличия) сторожевого условия.

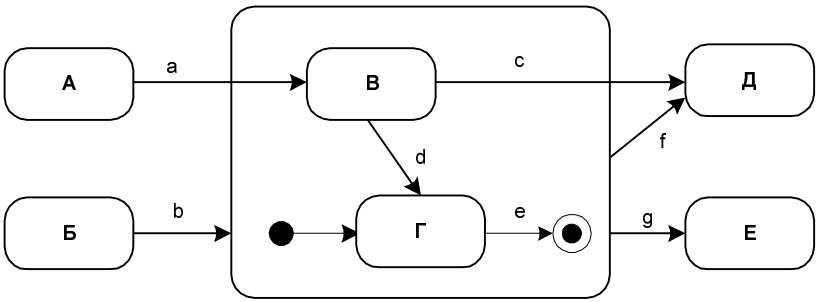


Рис. 8.12. Различные варианты переходов в составное состояние и из составного состояния

Иногда желательно реализовать ситуацию, когда выход из отдельного вложенного подсостояния соответствовал бы выходу и из составного состояния тоже. В этом случае изображают переход, который непосредственно выходит из вложенного подсостояния и пересекает границу суперсостояния (переход *c* на рис. 8.12). Аналогично, допускается изображение переходов, входящих извне составного состояния в отдельное вложенное состояние (переход *a* на рис. 8.12).

8.6.3. Синхронизирующие состояния

Как уже было отмечено, поведение параллельных конечных подавтоматов происходит независимо друг от друга, что позволяет, например, моделировать многозадачность в программных системах. Однако в отдельных ситуациях может возникнуть необходимость учесть в модели синхронизацию наступления отдельных событий и срабатывание соответствующих переходов. Для этой цели в языке UML имеется специальное псевдосостояние, которое называется синхронизирующим состоянием.

Синхронизирующее состояние (synch state) обозначается небольшой окружностью, внутри которой помещен символ звездочки "*". Оно используется совместно с переходом-слиянием или переходом-разделением для того, чтобы явно указать события в других конечных подавтоматах, оказывающие непосредственное влияние на поведение данного подавтомата.

Для иллюстрации использования синхронизирующих состояний рассмотрим упрощенную ситуацию с моделированием процесса постройки дома. Предположим, что постройка дома включает в себя строительные работы (возве-

дение фундамента и стен, возведение крыши и отделочные работы) и работы по электрификации дома (подведение электрической линии, прокладка скрытой электропроводки и установка осветительных ламп). Очевидно, два этих комплекса работ могут выполняться параллельно, однако между ними есть некоторая взаимосвязь.

В частности, прокладка скрытой электропроводки может начаться лишь после того, как будет завершено возведение фундамента и стен. Отделочные работы следует начать лишь после того, как будет закончена прокладка скрытой электропроводки. В противном случае отделочные работы придется проводить повторно. Рассмотренные особенности синхронизации этих параллельных процессов учтены на соответствующей диаграмме состояний с помощью двух синхронизирующих состояний (рис. 8.13).

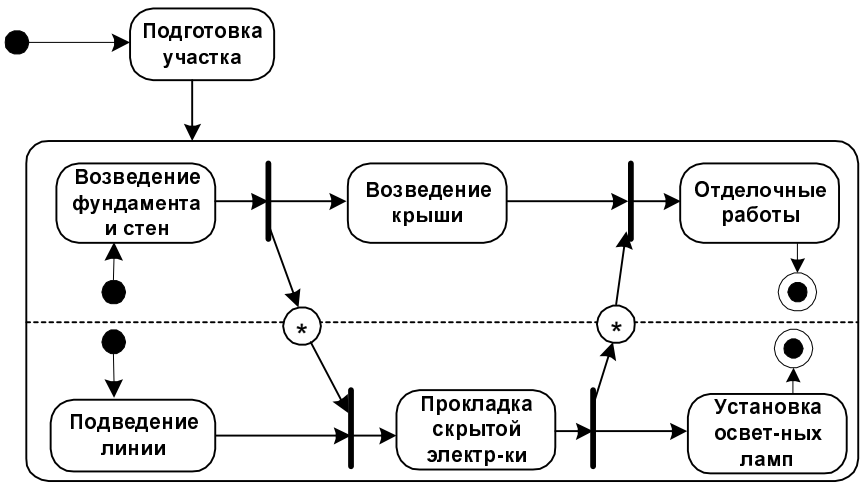


Рис. 8.13. Диаграмма состояний для примера со строительством дома

В завершение этого подраздела рассмотрим диаграмму состояний, которая представляет собой пример моделирования поведения конкретной системы — процесса функционирования телефонного аппарата (рис. 8.14). Этот пример иллюстрирует все основные особенности графической нотации, используемой при построении диаграммы состояний.

Кратко прокомментируем основные особенности этого примера. Данная диаграмма состояний представляет единственный конечный автомат с одним составным состоянием `Дозвон и разговор с абонентом`. Вне этого составного состояния имеется только одно состояние `Ожидание`, которое характеризует исправный и подключенный к телефонной сети телефонный аппарат. Переход в следующее состояние происходит при поднятии телефонной трубки. Переход с атомарным действием `подать тон-сигнал` переводит аппарат в составное состояние, а точнее — в его начальное подсостояние.

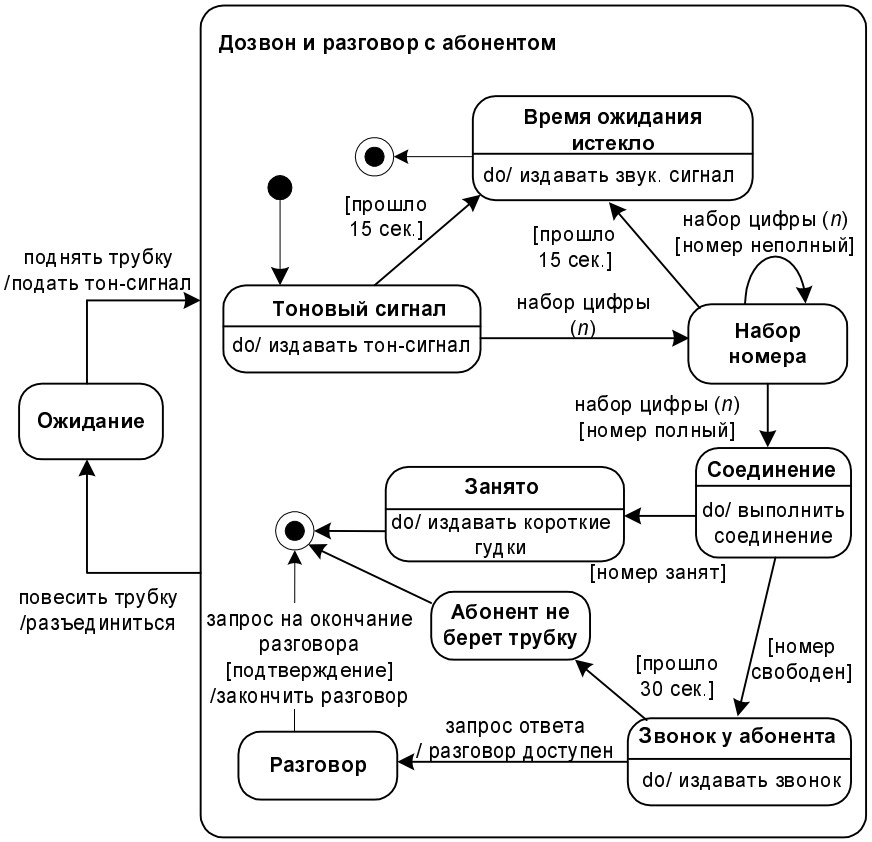


Рис. 8.14. Диаграмма состояний процесса функционирования телефонного аппарата

Далее телефонный аппарат сразу переходит в подсостояние `Тоновый сигнал`. При этом будет непрерывно издаваться тоновый сигнал до тех пор, пока не произойдет либо событие-триггер `набор цифры(n)`, либо не истечет 15 секунд с момента поднятия трубки. В первом случае аппарат перейдет в состояние `Набор номера`, а во втором — в состояние `Время ожидания истекло`. Последняя ситуация может быть результатом сомнений по поводу "звонить — не звонить?", следствием чего могут быть гудки в трубке. При этом нам ничего не остается делать, как опустить ее на рычаг.

При наборе номера выполняется событие-триггер `набор цифры(n)` со сторожевым условием `номер неполный`. Это означает, что если набранный телефонный номер не содержит необходимого количества цифр, то нам следует продолжать набор очередной цифры, находясь в состоянии `Набор номера`.

Если же набранный номер полный, то можно перейти в состояние `Соединение`. В результате соединения может оказаться, что аппарат абонента занят

(переход в состояние `Занято`) или свободен (переход в состояние `Звонок у абонента`). В первом случае можно повторить дозвон, предварительно опустив трубку на рычаг (выход из составного состояния). Во втором случае телефон у вызываемого абонента будет звонить до тех пор, пока им не будет поднята трубка или не пройдет интервал времени, равный 30 сек. Очевидно, в последнем случае, в контексте рассматриваемой модели нам ничего не остается, как опустить трубку на рычаг.

Если же вызываемый абонент поднял трубку, тем самым инициируется событие `запрос ответа` и происходит переход в состояние `Разговор` с выполнением действия `разговор доступен`. В состоянии `Разговор` моделируемая система может находиться как угодно долго. Для выхода из него необходимо выполнение триггерного перехода `запрос на окончание разговора` со стороны одного из абонентов и если разговаривают интеллигентные люди, то и взаимное согласие закончить разговор. Последнее моделируется с помощью сторожевого условия: `подтверждение`. Таким образом, после слов прощания и выполнения этого перехода разговор заканчивается, и мы опускаем трубку. При этом телефонный аппарат переходит в состояние `Ожидание`, в котором может находиться неопределенно долго.

Примечание

Строго говоря, рассмотренный пример отражает не все аспекты поведения даже такого несложного на первый взгляд устройства, каким является обычный телефонный аппарат. В случае набора полного номера нас могут ошибочно соединить с другим абонентом. При этом нам ничего не остается, как покинуть составное состояние, опустив трубку на рычаг. Другая модификация может быть связана с желанием повторно использовать набранный номер в случае коротких гудков "занято" у вызываемого абонента. Решение этой задачи может быть реализовано на основе использования исторического состояния вместо начального состояния для суперсостояния `Дозвон и разговор с абонентом`, которое будет запоминать в памяти аппарата набранный ранее номер. Можно также ограничить пребывание моделируемой системы в подсостоянии `Разговор`, чтобы предотвратить бесконечные разговоры потенциальных соседей.

Дополнить данную диаграмму состояний читателю предлагается самостоятельно в качестве упражнения.

8.7. Пример построения диаграммы состояний системы управления банкоматом

Продолжая построение модели сквозного примера модели системы управления банкоматом, построим диаграмму состояний для моделирования процесса функционирования банкомата как системы в целом. В этом случае в рамках разрабатываемой модели нам достаточно построить единственную диаграмму состояний, которая будет описывать реакцию банкомата на все

события при реализации всех вариантов использования. Вариант диаграммы состояний системы управления банкоматом изображен на рис. 8.15.

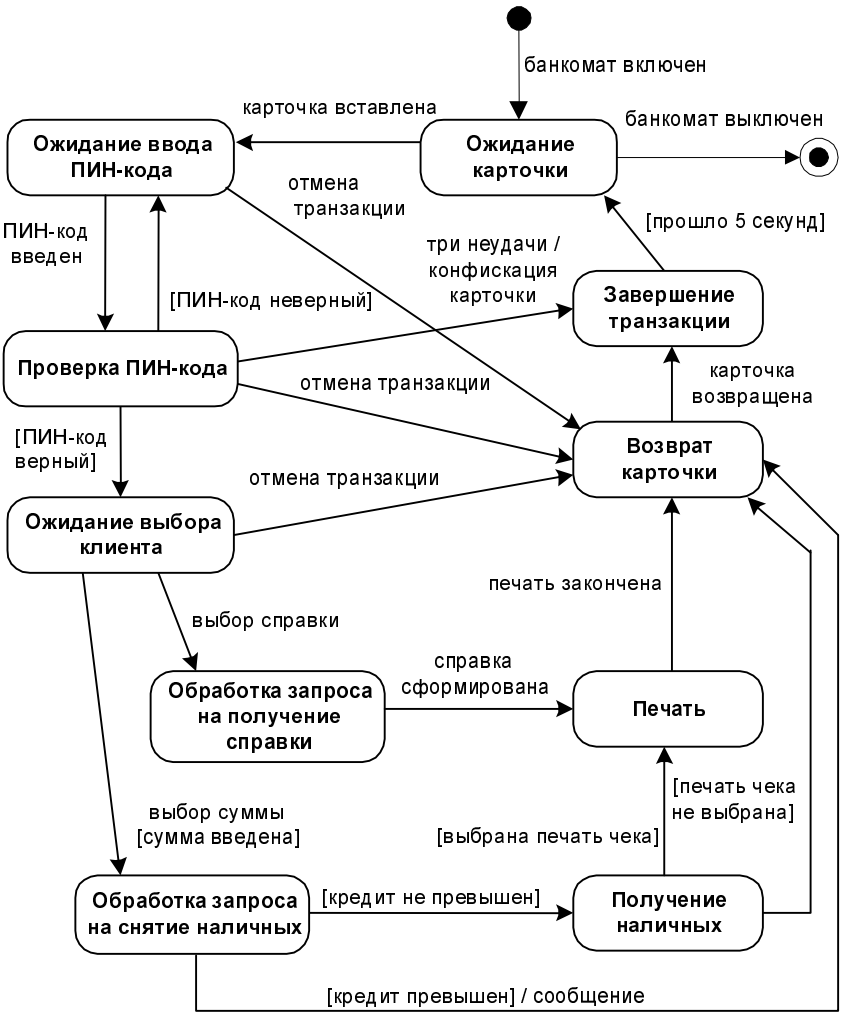


Рис. 8.15. Диаграмма состояний системы управления банкоматом

Данная диаграмма содержит 12 состояний, два из которых (начальное и конечное) являются псевдосостояниями. Для простоты в качестве имен некоторых состояний используется спецификация деятельности, которая выполняется в процессе нахождения банкомата в соответствующих состояниях. Для подобных состояний (например, `Проверка ПИН-кода`, `Получение наличных`, `Завершение транзакции`) выходными переходами являются нетриггерные переходы, для которых указаны только сторожевые условия.

Примечание

При внимательном рассмотрении построенных диаграмм состояний может обнаружиться ситуация, которая имеет место в действительности, но не нашла отражения на диаграмме, а значит — и в модели в целом. Применительно к системе управления банкоматом такой ситуацией может быть отсутствие требуемой суммы наличных в самом банкомате. Внести соответствующие дополнения в модель предлагается читателям самостоятельно в качестве упражнения.

Достоинством рассмотренной диаграммы состояний является возможность визуализировать на одном рабочем листе модели процесс поведения рассматриваемой системы в целом. Таким образом, полная модель системы может содержать либо единственную диаграмму состояний, описывающую реализации всех специфицированных вариантов использования (типичный ход событий и все исключения), либо несколько диаграмм состояний для отдельных ее подсистем (объектов). В последнем случае каждая из подсистем (объектов) должна иметь нетривиальное поведение, которое затрудняет его визуализацию на одной диаграмме состояний и служит веским аргументом построения нескольких диаграмм этого типа.

8.8. Заключительные рекомендации по построению диаграмм состояний

Основные особенности построения диаграмм состояний были рассмотрены при описании соответствующих модельных элементов. Однако некоторые моменты не нашли отражения, о чем необходимо сказать в заключение этой главы.

По своему назначению диаграмма состояний не является обязательным представлением в модели и как бы "присоединяется" к тому элементу, который, по замыслу разработчиков, имеет нетривиальное поведение в течение своего жизненного цикла. Наличие у системы нескольких состояний, отличающихся от простой дихотомии "исправен — неисправен", "активен — неактивен", "ожидание — реакция на внешние действия", уже служит признаком необходимости построения диаграммы состояний. В качестве начального варианта диаграммы состояний, если нет очевидных соображений по поводу состояний объекта, можно воспользоваться подобными состояниями, рассматривая их как составные и уточняя их (детализируя их внутреннюю структуру) по мере рассмотрения логики поведения моделируемой системы или объекта.

При выделении состояний и переходов следует помнить, что длительность срабатывания отдельных переходов должна быть существенно меньше, чем нахождение моделируемых элементов в соответствующих состояниях.

Каждое из состояний должно характеризоваться определенной устойчивостью во времени. Другими словами, из каждого состояния на диаграмме не может быть самопроизвольного перехода в какое бы то ни было другое состояние. Все переходы должны быть явно специфицированы, в противном случае построенная диаграмма состояний является либо неполной (неадекватной), либо ошибочной с точки зрения нотации языка UML (ill formed).

При разработке диаграммы состояний нужно постоянно следить, чтобы объект в каждый момент находился только в единственном состоянии. Если это не так, то данное обстоятельство может быть следствием ошибки или неявным признаком наличия параллельности у поведения моделируемого объекта. В последнем случае следует явно специфицировать необходимое число конечных подавтоматов, вложив их в то составное состояние, которое характеризуется нарушением условия одновременности.

Следует выполнять обязательную проверку того, чтобы никакие два перехода из одного состояния не могли сработать одновременно (требование отсутствия конфликтов у переходов). Наличие такого конфликта может служить признаком ошибки или неявной параллельности типа ветвления рассматриваемого процесса на два и более подконечных автомата. Если параллельность по замыслу разработчика отсутствует, то следует ввести дополнительные сторожевые условия либо изменить существующие, чтобы исключить конфликт переходов. При наличии параллельности следует заменить конфликтующие переходы одним параллельным переходом типа ветвления.

Использование исторических состояний оправдано в том случае, когда необходимо организовать обработку исключительных ситуаций (прерываний) без потери данных или выполненной работы. При этом применять исторические состояния, особенно глубокие, необходимо с известной долей осторожности. Нужно помнить, что каждый из конечных подавтоматов может иметь только одно историческое состояние. В противном случае возможны ошибки, особенно, когда подавтоматы изображаются на отдельных диаграммах состояний.

И, наконец, следует отметить, что некоторые дополнительные конструкции конечных автоматов, такие как точки динамического выбора (dynamic choice points) или точки соединения (junction points), в книге не нашли отражения. Это сделано по той причине, что данные модельные элементы хотя и позволяют моделировать более сложные аспекты динамического управления поведением объекта, но не являются базовыми. Соответствующая информация содержится в оригинальной документации по языку UML.

В заключение приводится сводка всех рассмотренных графических обозначений, которые могут быть использованы при построении диаграмм состояний (табл. 8.1).

Таблица 8.1. *Графические элементы диаграмм состояний*

Графическое изображение	Название
Имя состояния	Состояние (state)
Имя состояния entry / действие на входе do / деятельность exit / действие на выходе	Состояние с внутренними действиями и деятельностью
	Переход (transition)
	Начальное состояние
	Конечное состояние
H	Неглубокое историческое состояние
H*	Глубокое историческое состояние
*	Синхронизирующее состояние
	Разделение (fork)
	Слияние (join)
Имя состояния	Рефлексивный переход или переход в себя

Глава 9

Диаграмма деятельности (activity diagram)

При моделировании поведения проектируемой или анализируемой системы возникает необходимость не только представить процесс изменения ее состояний, но и детализировать особенности алгоритмической и процедурной реализации выполняемых системой операций. Традиционно для этой цели использовались блок-схемы или структурные схемы алгоритмов (см. рис. 1.1). Каждая такая схема акцентирует внимание на последовательности выполнения определенных действий или элементарных операций, которые в совокупности приводят к получению желаемого результата.

С алгоритмическими и логическими операциями, требующими своего выполнения в определенной последовательности, мы сталкиваемся в самых различных бытовых и деловых ситуациях. Конечно, мы не всегда задумываемся о том, что подобные операции относятся к столь научным категориям. Например, чтобы позвонить по телефону, нам предварительно нужно снять трубку и убедиться, что телефон подключен к линии. Для приготовления кофе или заваривания чая необходимо вначале вскипятить воду. Чтобы выполнить ремонт двигателя автомобиля, требуется осуществить целый ряд нетривиальных операций, таких как разборка силового агрегата, снятие генератора и некоторых других.

Важно подчеркнуть то обстоятельство, что с увеличением сложности системы строгое соблюдение определенной последовательности выполняемых действий приобретает все более важное значение. Если попытаться заварить кофе холодной водой, то мы можем только испортить одну порцию напитка. Нарушение последовательности операций при ремонте двигателя может привести к его поломке или выходу из строя. Еще более катастрофические последствия могут произойти в случае отклонения от установленной последовательности действий при взлете или посадке авиалайнера, запуске ракеты, регламентных работах на АЭС.

Для моделирования процесса выполнения операций в языке UML используются так называемые *диаграммы деятельности*. Применяемая в них графическая нотация во многом похожа на нотацию диаграммы состояний, поскольку на диаграммах деятельности также присутствуют обозначения

состояний и переходов. Отличие заключается в семантике состояний, которые используются для представления деятельности и действий, и в отсутствии на переходах сигнатуры событий. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении операции в предыдущем состоянии. Графически диаграмма деятельности представляется в форме графа деятельности, вершинами которого являются состояния действия, а дугами — переходы от одного состояния действия к другому.

Таким образом, диаграммы деятельности можно считать частным случаем диаграмм состояний. Именно они позволяют реализовать в языке UML особенности процедурного и синхронного управления, обусловленного завершением внутренних действий и деятельности. Метамодель UML предоставляет для этого необходимые термины и семантику. Основным направлением использования диаграмм деятельности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения. При этом каждое состояние может являться выполнением операции некоторого класса либо ее части, позволяя использовать диаграммы деятельности для описания реакций на внутренние события системы.

В контексте языка UML *деятельность* (activity) представляет собой некоторую совокупность отдельных вычислений, выполняемых автоматом. При этом отдельные элементарные вычисления могут приводить к некоторому результату или действию (action). На диаграмме деятельности отображается логика или последовательность перехода от одной деятельности к другой, при этом внимание фиксируется на результате деятельности. Сам же результат может привести к изменению состояния системы или возвращению некоторого значения.

Примечание

Хотя диаграмма деятельности предназначена для моделирования поведения систем, время в явном виде отсутствует на этой диаграмме. Ситуация здесь во многом аналогична диаграмме состояний.

9.1. Состояние действия

Состояние действия (action state) является специальным случаем состояния с некоторым входным действием и, по крайней мере, одним выходящим из состояния переходом. Этот переход неявно предполагает, что входное действие уже завершилось. Состояние действия не может иметь внутренних переходов, поскольку оно является элементарным. Обычное использование состояния действия заключается в моделировании одного шага выполнения алгоритма (процедуры) или потока управления.

Графически состояние действия изображается фигурой, напоминающей прямоугольник со сферическими боковыми сторонами (рис. 9.1). Внутри этой фигуры записывается имя состояния действия в форме *выражение действия* (action-expression), которое должно быть уникальным в пределах одной диаграммы деятельности.

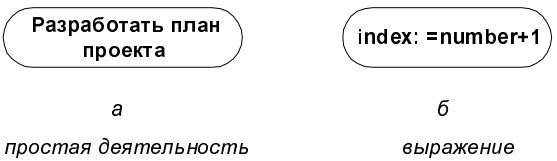


Рис. 9.1. Графическое изображение состояния действия

Действие может быть записано на естественном языке, некотором псевдокоде или языке программирования. Никаких дополнительных или неявных ограничений при записи действий не накладывается. Рекомендуется в качестве имени простого действия использовать глагол с пояснительными словами (рис. 9.1, *а*). Если же действие может быть представлено в некотором формальном виде, то целесообразно записать его на том языке программирования, на котором предполагается реализовывать разрабатываемый проект (рис. 9.1, *б*).

Иногда возникает необходимость представить на диаграмме деятельности некоторое сложное действие, состоящее из нескольких более простых. В этом случае можно использовать специальное обозначение так называемого *состояния под-деятельности* (subactivity state). Это состояние является графом деятельности и обозначается специальной пиктограммой в правом нижнем углу символа состояния действия (рис. 9.2). Данная конструкция может применяться к любому элементу языка UML, который поддерживает "вложенность" своей структуры. При этом пиктограмма может быть дополнительно помечена типом вложенной структуры.

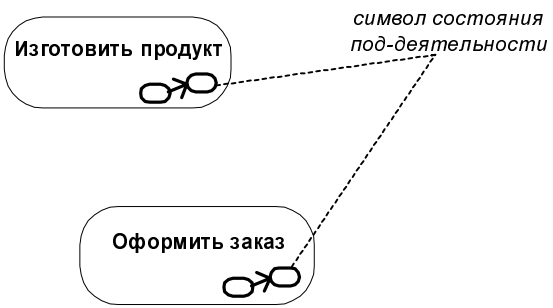


Рис. 9.2. Графическое изображение состояния под-деятельности

Каждая диаграмма деятельности должна иметь единственное начальное и конечное состояния. Они имеют такие же обозначения, как и на диаграмме состояний (см. рис. 6.4). При этом каждая деятельность начинается в начальном состоянии и заканчивается в конечном состоянии. Саму диаграмму деятельности принято располагать таким образом, чтобы действия следовали сверху вниз. В этом случае начальное состояние будет изображаться в верхней части диаграммы, а конечное — в ее нижней части.

Примечание

В интересах удобства визуального представления на диаграмме деятельности допускается изображать несколько конечных состояний. В этом случае все их принято считать эквивалентными друг другу.

9.2. Переходы

Переход как элемент языка UML был рассмотрен в *главе 8*. При построении диаграммы деятельности используются только нетриггерные переходы, т. е. те, которые срабатывают сразу после завершения деятельности или выполнения соответствующего действия состояния. Такой переход переводит деятельность в последующее состояние сразу, как только закончится действие в предыдущем состоянии. На диаграмме такой переход изображается сплошной линией со стрелкой (рис. 9.3).

Если из состояния действия выходит единственный переход, то он может быть никак не помечен. Если же таких переходов несколько, при моделировании последовательной деятельности может сработать только один из них. В этом случае для каждого из таких переходов должно быть явно записано собственное сторожевое условие в прямых скобках *(см. главу 8)*. При этом для всех выходящих из некоторого состояния переходов должно выполняться требование истинности только одного из них. Подобный случай встречается тогда, когда последовательно выполняемая деятельность должна разделиться на альтернативные ветви в зависимости от значения некоторого промежуточного результата. Такая ситуация получила название ветвления, а для ее обозначения применяется специальный символ решения.

Графически ветвление на диаграмме деятельности обозначается символом *решения* (decision), изображаемого в форме небольшого ромба, внутри которого нет никакого текста (см. рис. 9.3). В этот ромб может входить только одна стрелка от того состояния действия, после выполнения которого поток управления должен быть продолжен по одной из взаимно исключающих ветвей. Принято входящую стрелку присоединять к верхней или левой вершине символа решения. Выходящих стрелок может быть две или более, но для каждой из них явно указывается соответствующее сторожевое условие в форме булевского выражения.

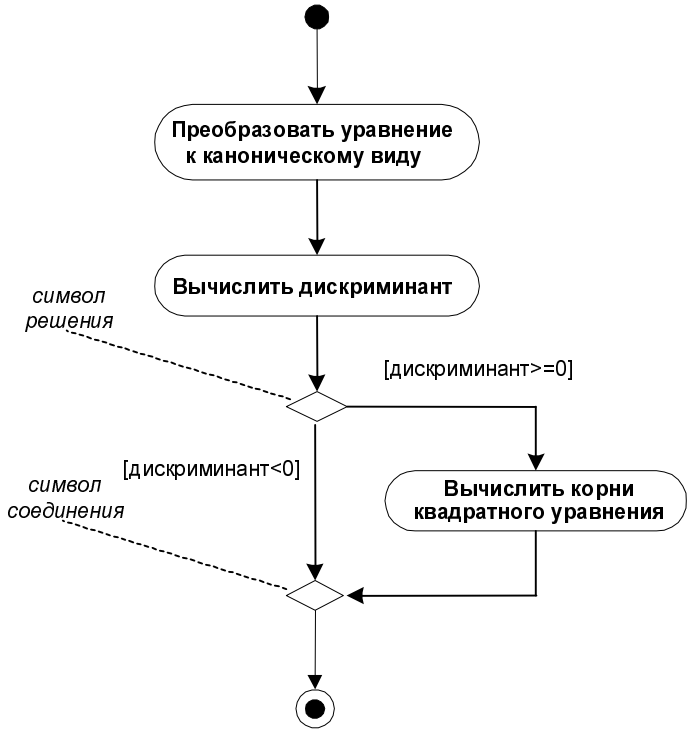


Рис. 9.3. Фрагмент диаграммы деятельности для алгоритма нахождения корней квадратного уравнения

Примечание

Для графического объединения альтернативных ветвей на диаграмме деятельности рекомендуется также использовать аналогичный символ в форме ромба, который в этом случае называют *соединением* (*merge*). Наличие этого символа, внутри которого также не записывается никакого текста, упрощает визуальный контроль логики выполнения процедурных действий на диаграмме деятельности (см. рис. 9.3). Входящих стрелок у символа соединения может быть несколько от тех состояний действия, каждое из которых принадлежит к одной из взаимно исключающих ветвей. Выходить из ромба может только одна стрелка, при этом ни входящие, ни выходящая стрелки не должны содержать сторожевых условий. Исключением является ситуация, когда с целью сокращения диаграммы объединяют символ решения с символом соединения. Нарушение этих правил делает диаграмму деятельности несостоятельной (ill formed).

В качестве примера рассмотрим построение диаграммы деятельности для алгоритма нахождения корней квадратного уравнения. В общем случае, после приведения уравнения второй степени к каноническому виду $ax^2 + bx + c = 0$, необходимо вычислить его дискриминант. Как следует из

школьного курса математики, в случае отрицательного значения дискриминанта квадратное уравнение не имеет решения на множестве действительных чисел, и дальнейшие вычисления должны быть прекращены. При неотрицательном значении дискриминанта уравнение имеет решение, корни которого могут быть получены на основе известной расчетной формулы.

Графически описанный алгоритм вычисления корней квадратного уравнения может быть представлен в виде фрагмента диаграммы деятельности с тремя состояниями действия, решением и соединением (рис. 9.3). Каждый из переходов, выходящих из символа решения после состояния `Вычислить дискриминант`, имеет сторожевое условие, которое определяет единственную ветвь, по которой может быть продолжен процесс вычисления корней в зависимости от знака дискриминанта. Очевидно, что в случае его отрицательности, мы сразу попадаем в конечное состояние, тем самым завершая выполнение алгоритма в целом.

В рассмотренном примере выполняемые альтернативно действия соединяются в символе соединения. Применительно к данному фрагменту это можно было бы изобразить иначе. А именно, каждую из ветвей завершить конечным состоянием, что в принципе будет эквивалентно изображенной диаграмме.

Примечание

Строго говоря, первое из состояний рассматриваемого алгоритма следует считать состоянием под-деятельности, поскольку приведение квадратного уравнения к каноническому виду может потребовать нескольких элементарных действий (приведение подобных и перенос отдельных членов уравнения из правой его части в левую). Поэтому для данного состояния целесообразно добавить соответствующую пиктограмму (см. рис. 9.2). Подставить соответствующие расчетные формулы в состояния диаграммы деятельности предлагается читателям самостоятельно в качестве упражнения.

В следующем примере (рис. 9.4) моделируется ситуация, встречающаяся в некоторых супермаркетах при оплате товаров по кредитной карточке. Если общая стоимость приобретаемых товаров превышает 50$, то выполняется аутентификация личности владельца карточки. В случае положительного результата проверки (карточка действительна), происходит снятие суммы со счета покупателя и оплата стоимости товаров. При отрицательном результате проверки (карточка недействительна) оплаты не происходит и товар остается у продавца.

Если общая стоимость приобретаемых товаров не превышает 50$, то аутентификация личности владельца карточки не выполняется. В этом случае происходит оплата предъявленной кредитной карточкой, после чего товар становится собственностью покупателя.

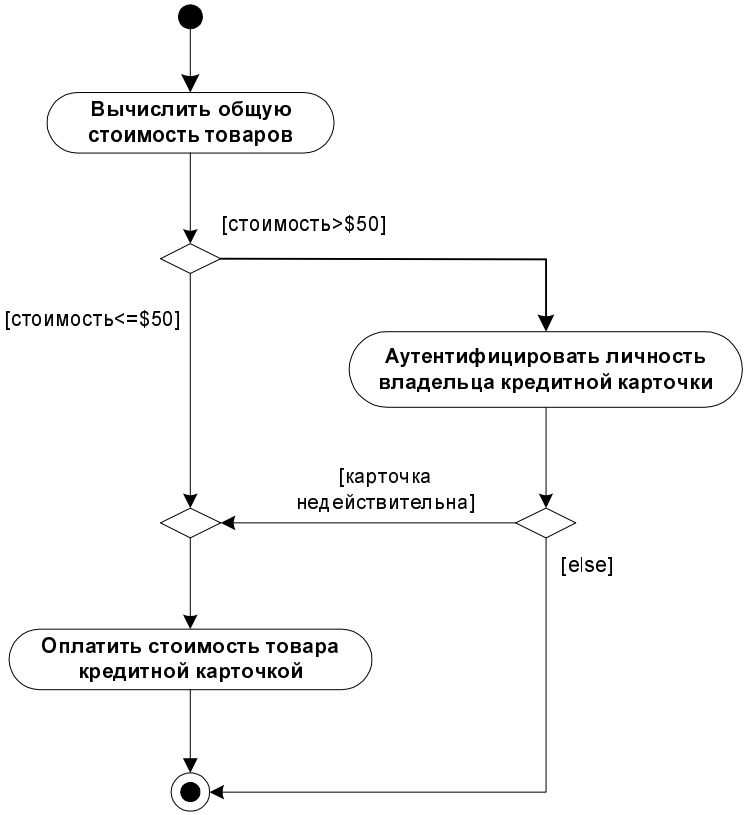


Рис. 9.4. Различные варианты ветвлений на диаграмме деятельности

Примечание

Как видно из данного рисунка, из соображений удобства допускается использовать вместо одного из сторожевых условий бинарного решения слово "иначе" (else), указывающее на тот переход, который должен сработать в случае невыполнения специфицированного явным образом сторожевого условия.

Один из наиболее значимых недостатков обычных блок-схем или структурных схем алгоритмов связан с проблемой изображения параллельных ветвей отдельных вычислений. Поскольку распараллеливание вычислений существенно повышает общее быстродействие программных систем, необходимы графические примитивы для представления параллельных процессов. Для диаграмм деятельности для этой цели используется специальный символ для разделения и слияния параллельных вычислений или потоков управления. Таким символом является прямая черточка, аналогично обозначению параллельных переходов для диаграмм состояний.

На диаграммах деятельности такая черточка изображается отрезком горизонтальной, реже — вертикальной линии, толщина которой несколько шире линий простых переходов диаграммы деятельности. При этом *разделение* (concurrent fork) имеет один входящий переход и несколько выходящих (рис. 9.5, *а*), которые изображаются отрезками вертикальных, реже — горизонтальных линий. *Слияние* (concurrent join), наоборот, имеет несколько входящих переходов и один выходящий (рис. 9.5, *б*). Особенностью параллельных переходов на диаграмме деятельности является возможность их изображения в удлиненной форме для возможности показать входящие и выходящие переходы вертикальными стрелками.

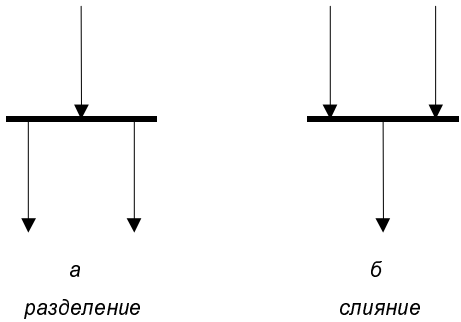


Рис. 9.5. Графическое изображение разделения и слияния параллельных потоков управления на диаграмме деятельности

Для иллюстрации особенности изображения ветвления и параллельных подпроцессов удобно рассмотреть пример с выбором напитка и приготовлением кофе с помощью кофеварки. Достоинство этого примера состоит в том, что он практически не требует никаких дополнительных пояснений в силу очевидности своего контекста (рис. 9.6).

Ветвление в рассматриваемом примере проявляется собственно в выборе напитка для утоления жажды, а параллельность — в том, что поисками чашки мы можем заняться во время приготовления кофе.

Примечание

Впрочем, выбор конкретных напитков, так же как и выбор кофеварки, остается за читателем. В этом случае разработка соответствующей диаграммы деятельности может быть им предложена в качестве упражнения.

Таким образом, диаграмма деятельности является специальным случаем диаграммы состояний, в которой все состояния являются состояниями действия или под-деятельности, а все переходы — нетриггерными переходами, т. е. переходами, которые происходят по завершении действий или поддеятельностей в отдельных состояниях.

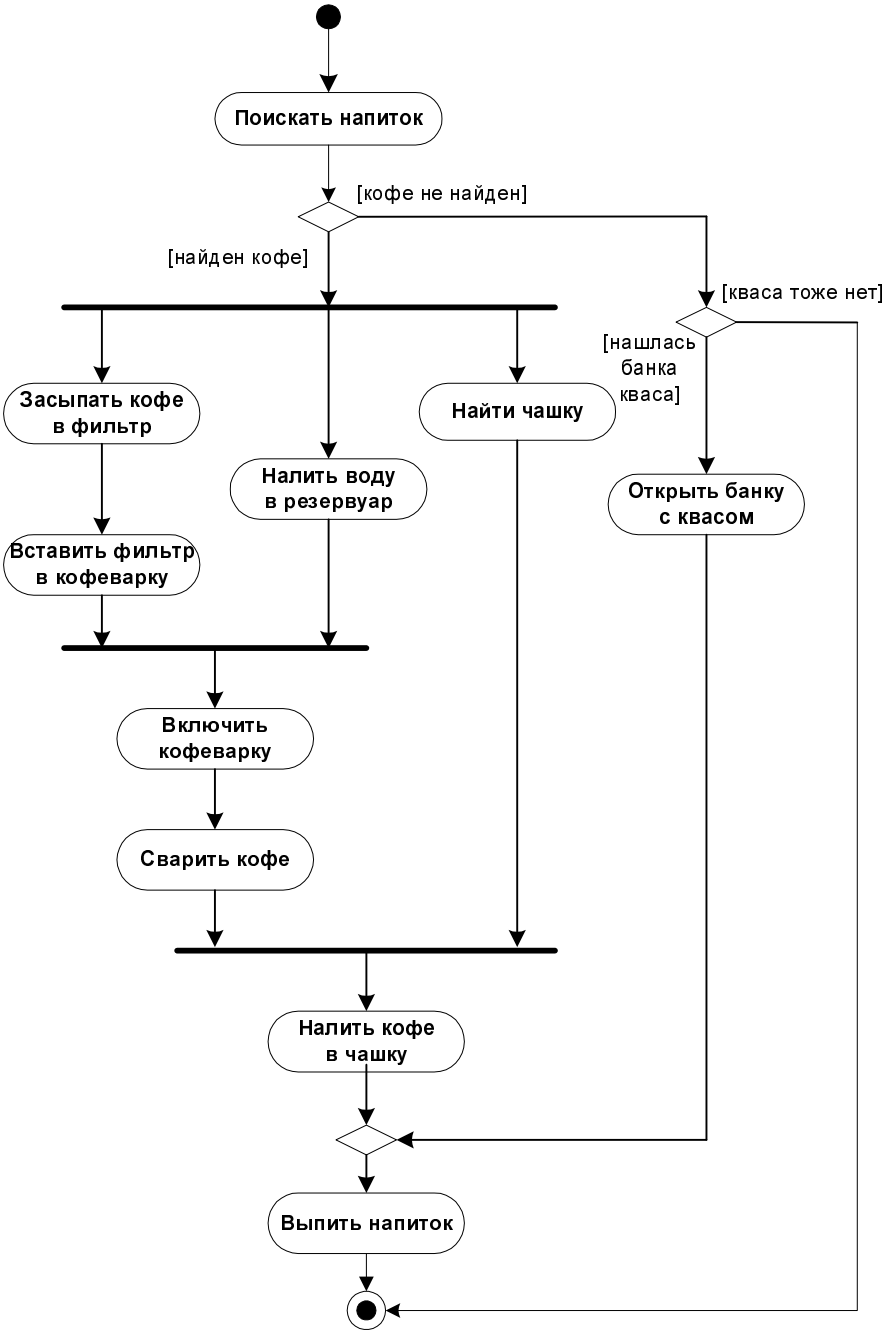


Рис. 9.6. Диаграмма деятельности для примера с приготовлением напитка

9.3. Дорожки

Диаграммы деятельности могут быть использованы не только для спецификации алгоритмов вычислений или потоков управления в программных системах. Не менее важная область их применения связана с моделированием бизнес-процессов. Действительно, деятельность любой компании (фирмы) представляет собой не что иное, как совокупность отдельных действий (работ, операций), направленных на достижение требуемого результата. Однако применительно к бизнес-процессам желательно выполнение каждого действия ассоциировать с конкретным подразделением компании. В этом случае соответствующее подразделение будет нести ответственность за реализацию отдельных действий, а сам бизнес-процесс представляется в виде переходов действий из одного подразделения к другому.

Для моделирования этих особенностей в языке UML предложена специальная конструкция, получившая название *дорожки* (swimlane). Имеется в виду визуальная аналогия с плавательными дорожками в бассейне, если смотреть на соответствующую диаграмму. При этом все состояния действия на диаграмме деятельности делятся на отдельные группы, которые отделяются друг от друга вертикальными линиями. Две соседних линии и образуют дорожку, а группа состояний между этими линиями выполняется отдельным организационным подразделением (отделом, группой, отделением, филиалом) или сотрудником компании (рис. 9.7). В последнем случае принято указывать должность сотрудника, ответственного за выполнение соответствующих действий.

Названия подразделений или должностей явно указываются в верхней части дорожки. Пересекать линию дорожки могут только переходы, которые в этом случае обозначают выход или вход потока управления в соответствующее подразделение компании. Порядок следования дорожек не несет какой-либо семантической информации и определяется соображениями удобства.

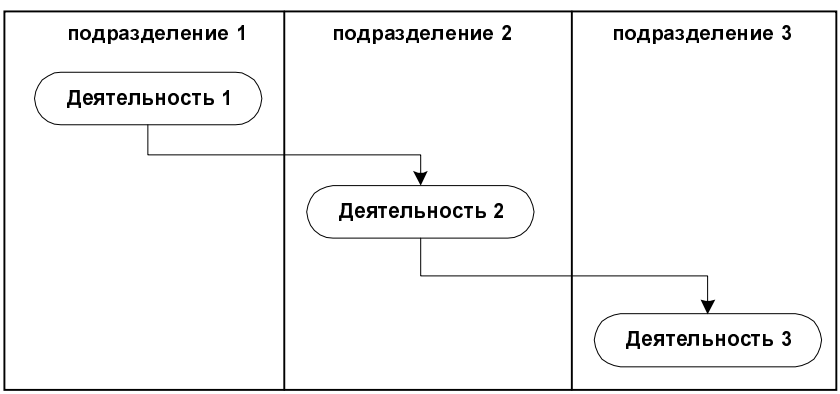


Рис. 9.7. Вариант диаграммы деятельности с дорожками

В качестве примера рассмотрим фрагмент диаграммы деятельности торговой компании, обслуживающей клиентов по телефону. Подразделениями компании обычно являются отдел приема и оформления заказов, отдел продаж и склад. Этим подразделениям будут соответствовать три дорожки на диаграмме деятельности, каждая из которых специфицирует зону ответственности подразделения. В этом случае диаграмма деятельности заключает в себе не только информацию о последовательности выполнения рабочих действий, но и о том, какое из подразделений торговой компании должно выполнять то или иное действие (рис. 9.8).

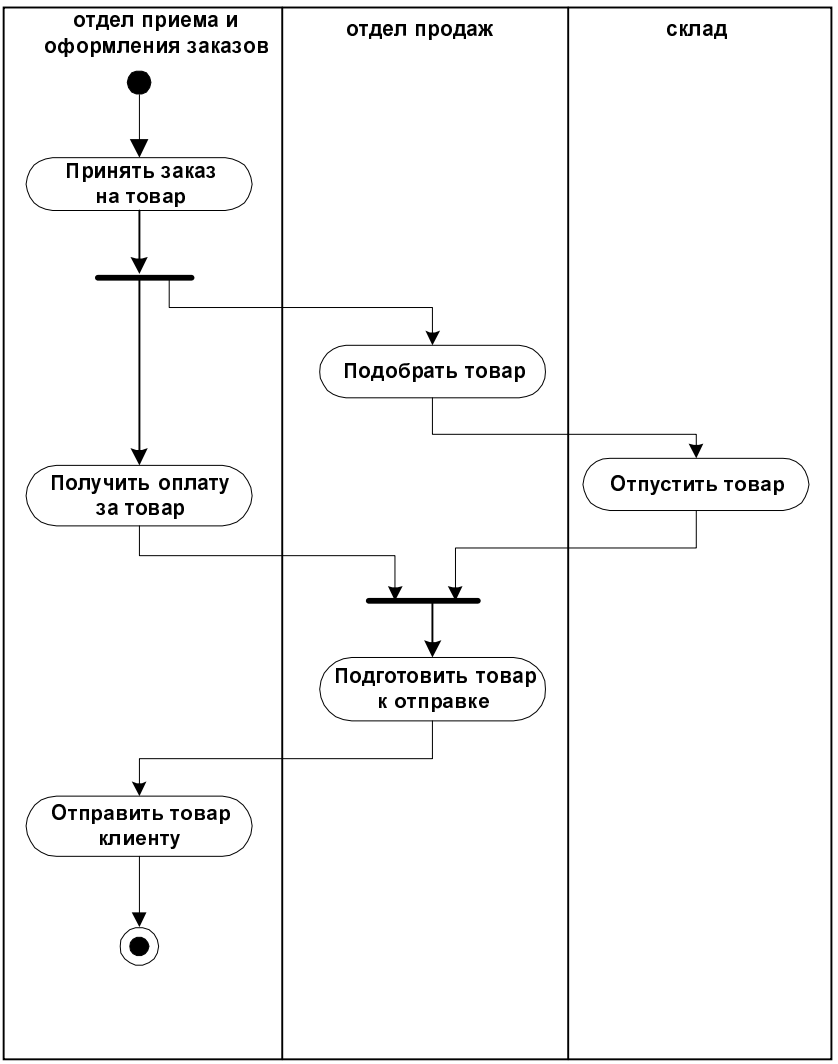


Рис. 9.8. Фрагмент диаграммы деятельности для торговой компании

Из указанной диаграммы деятельности сразу видно, что после принятия заказа от клиента отделом приема и оформления заказов осуществляется распараллеливание деятельности на два потока (переход-разделение). Первый из них остается в этом же отделе и связан с получением оплаты от клиента за заказанный товар. Второй инициирует выполнение действия по подбору товара в отделе продаж (модель товара, размеры, цвет, год выпуска и пр.). После окончания этой работы инициируется действие по выдаче товара со склада. Однако подготовка товара к отправке в торговом отделе начинается только после того, как будет получена оплата за товар от клиента и товар будет отпущен со склада (переход-слияние). Только после этого товар отправляется клиенту, переходя в его собственность.

9.4. Объекты

Действия на диаграмме деятельности могут выполняться над теми или иными объектами. Эти объекты либо инициируют выполнение действий, либо определяют некоторый результат этих действий. При этом действия специфицируют вызовы, которые передаются от одного объекта графа деятельности другому. Поскольку в таком ракурсе объекты играют определенную роль в понимании процесса деятельности, иногда возникает необходимость явно указать их на диаграмме деятельности.

Напомним, что базовым графическим представлением объекта в нотации языка UML является прямоугольник класса, с тем отличием, что имя объекта подчеркивается *(см. главы 6 и 7)*. На диаграммах деятельности после имени может указываться характеристика состояния объекта в прямых скобках. Такие прямоугольники объектов присоединяются к состояниям действия отношением зависимости пунктирной линией со стрелкой. Соответствующая зависимость определяет состояние конкретного объекта после выполнения предшествующего действия.

На диаграмме деятельности с дорожками расположение объекта может иметь некоторый дополнительный смысл. А именно, если объект расположен на границе двух дорожек, то это может означать, что переход к следующему состоянию действия в соседней дорожке ассоциирован с готовностью некоторого документа (объект в некотором состоянии). Если же объект целиком расположен внутри дорожки, то и состояние этого объекта целиком определяется действиями данной дорожки.

Возвращаясь к предыдущему примеру с торговой компанией, можно заметить, что центральным объектом процесса продажи является заказ или, вернее состояние его выполнения. Сначала, до звонка от клиента, заказ, как объект, отсутствует и возникает лишь после такого звонка. Однако этот заказ еще не заполнен до конца, поскольку требуется еще подобрать конкретный товар в отделе продаж. После его подготовки он передается на склад, где вместе с отпуском товара происходит дальнейшее оформление заказа.

Наконец, после получения подтверждения об оплате товара, соответствующие данные вносятся в заказ и он считается выполненным и закрытым.

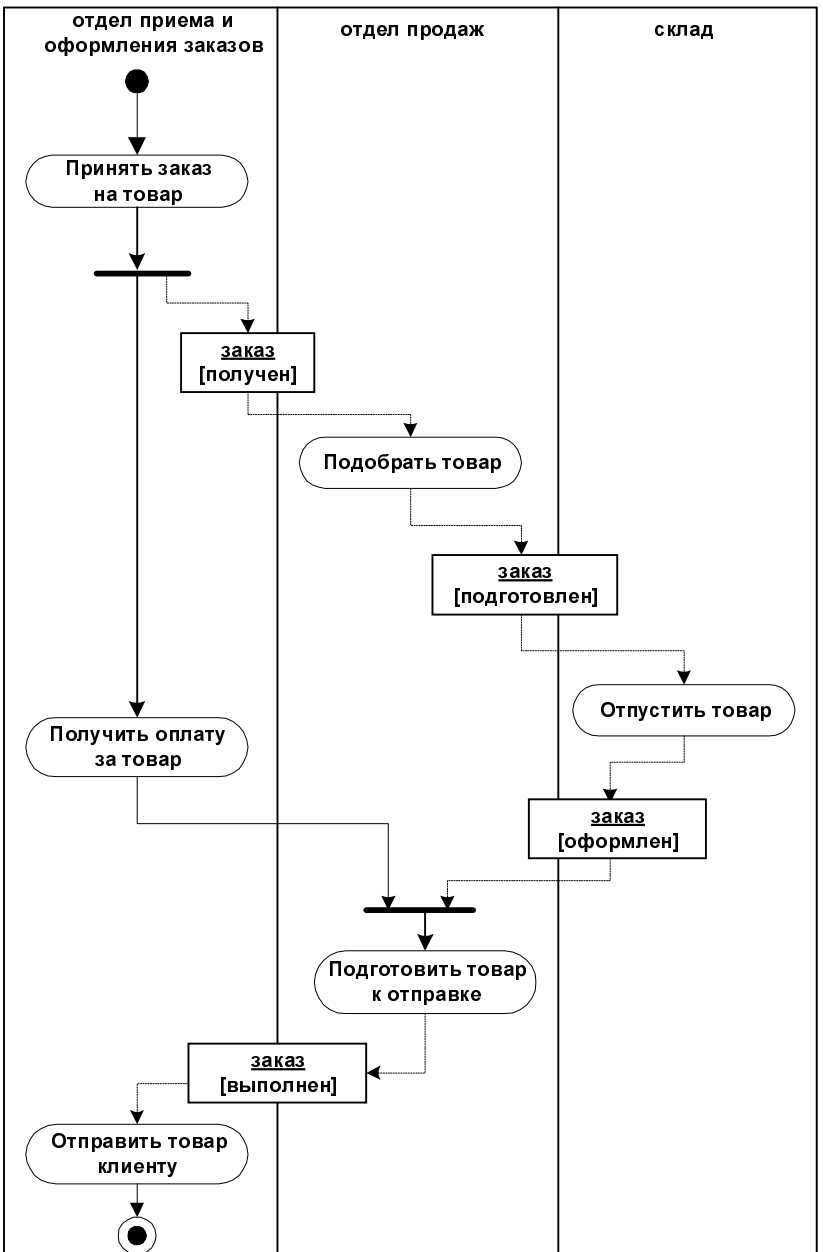


Рис. 9.9. Фрагмент диаграммы деятельности торговой компании с объектом-заказом

Эта информация может быть представлена графически в виде модифицированного варианта диаграммы деятельности этой же торговой компании (рис. 9.9).

Примечание

Применительно к диаграммам деятельности объекты, как правило, являются экземплярами классов сущностей или бизнес-сущностей. Следует также заметить, что на диаграмме деятельности один и тот же объект может быть изображен несколько раз, при этом для исключения несогласованности диаграммы необходимо указывать для них различные характеристики состояния.

В заключение следует остановиться на изображении синхронизации отдельных действий на диаграмме деятельности. Необходимость в согласовании действий возникает всякий раз, когда параллельно выполняемые действия оказывают влияние друг на друга. Если вспомнить материал предыдущей главы, то применительно к диаграмме состояний для этой цели применялось специальное псевдосостояние — синхронизирующее состояние. На диаграмме деятельности никаких дополнительных обозначений не используется, поскольку синхронизация параллельных процессов может быть реализована с помощью переходов "разделение-слияние".

В качестве примера рассмотрим упрощенную ситуацию с моделированием процесса постройки дома *(см. главу 8)*. Напомним, что в этом примере постройка дома включает в себя строительные работы (возведение фундамента и стен, возведение крыши и отделочные работы) и работы по электрификации дома (подведение электрической линии, прокладка скрытой электропроводки и установка осветительных ламп). Синхронизация параллельного выполнения этого комплекса работ может быть изображена на диаграмме деятельности следующим образом (рис. 9.10).

В рассмотренном примере все состояния являются состояниями под-деятельности. Это означает, что каждое из них можно детализировать в виде отдельного графа деятельности с соответствующей диаграммой. Действительно, состояние под-деятельности `Подготовка участка` может включать в себя такие действия, как очистка участка от деревьев, вывоз этих деревьев за пределы участка, рытье котлована под фундамент, установка временных строений для складирования строительных материалов и другие работы. Впрочем, некоторые из состояний могут быть и простыми состояниями действия в том случае, если соответствующая работа в рассматриваемом контексте является выполнением действия, неразложимого на отдельные элементарные операции или физические приемы.

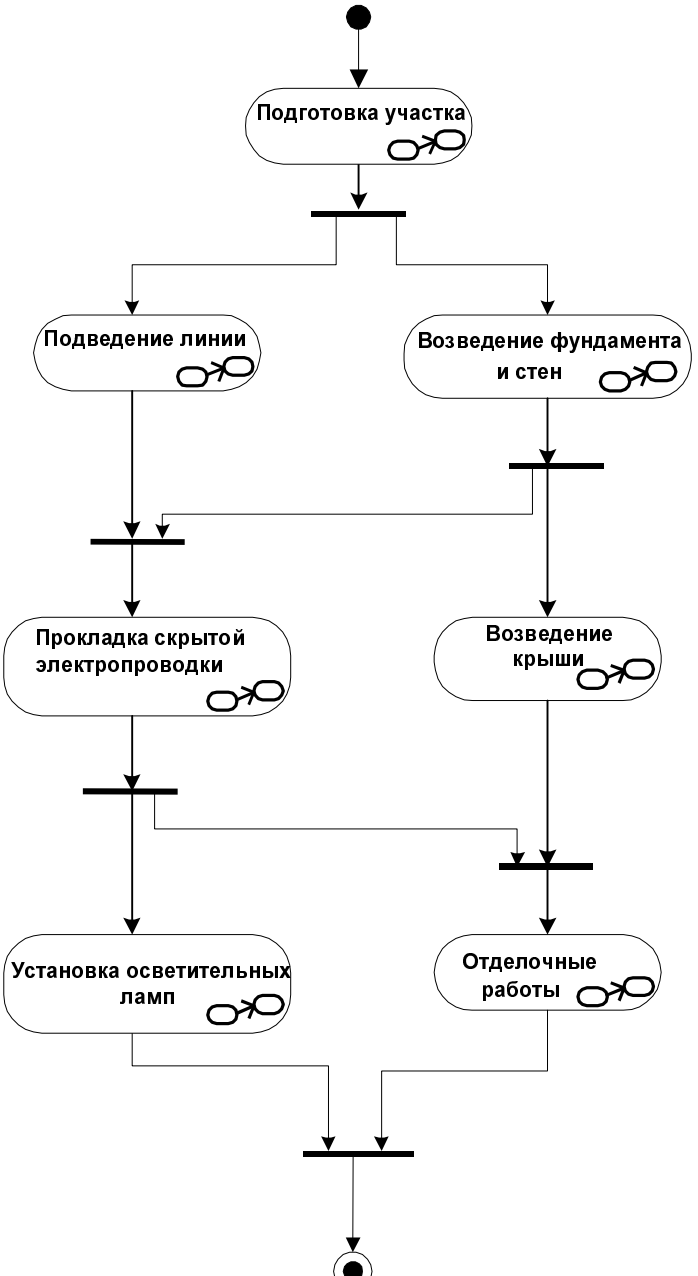


Рис. 9.10. Диаграмма деятельности с синхронизацией параллельных действий

9.5. Пример построения диаграммы деятельности системы управления банкоматом

Продолжая рассматривать сквозной пример модели системы управления банкоматом, построим диаграмму деятельности для графического представления процесса функционирования банкомата. В рамках разрабатываемой модели может быть построена диаграмма деятельности для реализации варианта использования `Снятие наличных по кредитной карточке`, на которой удобно представить последовательность действий банкомата наряду с проверкой ПИН-кода. Вариант диаграммы деятельности системы управления банкоматом для этого случая изображен на рис. 9.11.

Данная диаграмма содержит 6 состояний действий, начальное и конечное состояния, а также 4 символа решения и 4 символа соединения. В качестве имен всех состояний используется спецификация действий, которые в них выполняются. При этом все переходы на диаграмме деятельности являются нетриггерными, которые происходят по завершении выполнения соответствующих действий.

Примечание

Внимательные читатели обнаружат, что построенная диаграмма деятельности визуализирует ситуацию, когда в банкомате отсутствует требуемая сумма снимаемых наличных. Однако ни ситуация с печатью чека, ни реакция системы на троекратный неверный ввод ПИН-кода не нашла отражения на данной диаграмме. Внести соответствующие дополнения в модель предлагается читателям самостоятельно в качестве упражнения.

Достоинством диаграммы деятельности является возможность визуализировать отдельные аспекты поведения рассматриваемой системы или реализации отдельных операций классов в виде некоторой процедурной последовательности действий. Таким образом, полная модель системы может содержать одну или несколько диаграмм деятельности, каждая из которых описывает последовательность реализации либо наиболее важных вариантов использования (типичный ход событий и все исключения), либо нетривиальных операций классов.

Следует также отметить, что диаграмма деятельности с дорожками в контексте языка UML является основным средством визуализации и документирования бизнес-процессов. В этом случае каждый из бизнес-процессов рассматриваемой компании может быть изображен в форме отдельной диаграммы деятельности. Однако язык UML не содержит никаких дополнительных средств для имитационного моделирования (simulation) бизнес-процессов, что несколько ограничивает возможности его использования в бизнес-моделировании и реинжиниринге бизнес-процессов.

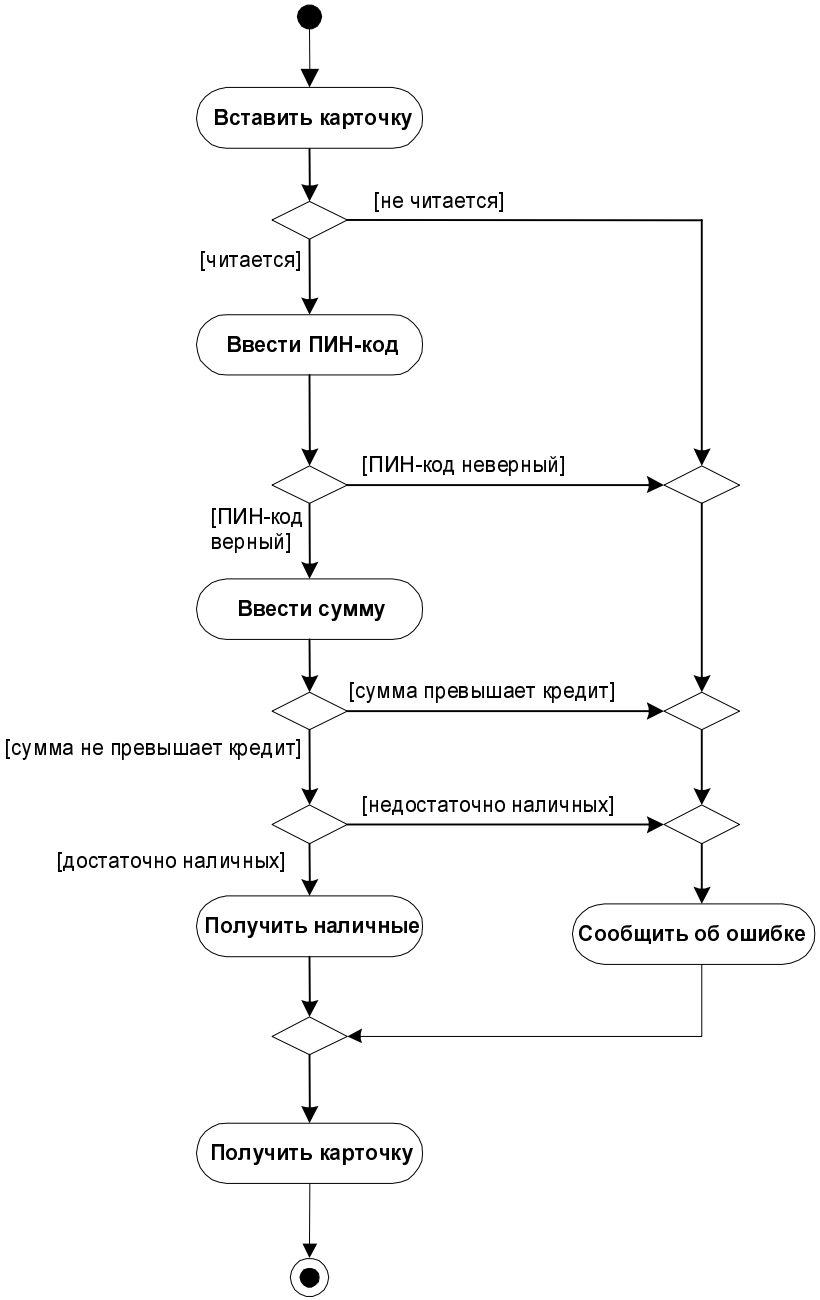


Рис. 9.11. Диаграмма деятельности процесса функционирования системы управления банкоматом

9.6. Рекомендации по построению диаграмм деятельности

Диаграммы деятельности играют важную роль в понимании процессов реализации алгоритмов выполнения операций классов и процедурных потоков управления в моделируемой системе. Используемые для этой цели традиционные блок-схемы алгоритмов обладают серьезными ограничениями в представлении параллельных процессов и их синхронизации. Применение дорожек и объектов открывает дополнительные возможности для наглядного представления бизнес-процессов, позволяя специфицировать деятельность подразделений коммерческих компаний и фирм.

Содержание диаграммы деятельности во многом напоминает диаграмму состояний, хотя и не тождественно ей. Поэтому многие рекомендации по построению последней оказываются справедливыми применительно к диаграмме деятельности. В частности, диаграмма деятельности может быть построена для объектов отдельных классов, варианта использования, отдельной операции класса или моделируемой системы в целом.

С одной стороны, на начальных этапах проектирования, когда детали реализации деятельности в проектируемой системе неизвестны, построение диаграммы деятельности начинают с выделения под-деятельностей, которые в совокупности образуют исходную деятельность системы. В последующем, по мере разработки диаграмм классов и состояний, эти под-деятельности уточняются в виде отдельных вложенных диаграмм деятельности для подсистем и объектов.

С другой стороны, отдельные участки рабочего процесса в существующей системе могут быть хорошо отлаженными, и у разработчиков может возникнуть желание сохранить этот механизм выполнения действий в проектируемой системе. Тогда строится диаграмма деятельности для этих участков, отражающая конкретные особенности выполнения действий с использованием дорожек и потока объектов. В последующем такая диаграмма вкладывается в более общие диаграммы деятельности для подсистемы и системы в целом, сохраняя свой уровень детализации.

В случае типового проекта большинство деталей реализации действий могут быть известны заранее на основе анализа существующих систем или предшествующего опыта разработки систем-прототипов. Для этой ситуации доминирующим будет восходящий процесс разработки. Использование типовых решений может существенно сократить время разработки и избежать возможных ошибок при реализации проекта.

При разработке проекта новой системы, процесс функционирования которой основан на новых технологических решениях, ситуация представляется более сложной. А именно, до начала работы над проектом могут быть неизвестны не только детали реализации отдельных действий, но и само содер-

жание деятельности становится предметом разработки. В этом случае доминирующим будет нисходящий процесс разработки от более общих схем к уточняющим их диаграммам. При этом достижение такого уровня детализации всех диаграмм, который достаточен для понимания особенностей реализации всех действий и под-деятельностей, может служить признаком завершения отдельных этапов работы над проектом.

В заключение следует заметить, что диаграмма деятельности, так же как и другие виды канонических диаграмм, не содержит средств выбора оптимальных вариантов конфигурации собственно диаграмм. При разработке сложных проектов проблема выбора оптимальных решений применительно к диаграммам деятельности становится весьма актуальной. Рациональное расходование средств, затраченных на разработку и эксплуатацию системы, повышение ее производительности и надежности зачастую определяют конечный результат всего проекта. В такой ситуации можно рекомендовать использование специализированных инструментальных средств, ориентированных на аналитико-имитационное исследование моделей системы на этапе разработки ее проекта.

В частности, при построении диаграмм деятельности сложных систем могут быть успешно использованы различные классы сетей Петри (классические, логико-алгебраические, стохастические, нечеткие и другие) и нейронных сетей. Применение этих формализмов позволяет на основе анализа модели системы не только получить оптимальную структуру ее поведения, но и специфицировать целый ряд дополнительных характеристик системы, которые не могут быть представлены на диаграмме деятельности и других диаграммах UML.

Таким образом, процесс объектно-ориентированного анализа и проектирования сложных систем представляется как последовательность итераций нисходящей и восходящей разработки отдельных диаграмм, включая и диаграмму деятельности. Доминирование того или иного из направлений разработки определяется особенностями конкретного проекта, его сложностью, масштабностью и новизной.

В заключение приводится сводка рассмотренных графических обозначений, которые используются при построении диаграмм деятельности (табл. 9.1). Хотя в языке UML имеются и некоторые другие элементы, данного перечня может оказаться вполне достаточно для выполнения большинства реальных проектов.

Таблица 9.1. *Графические элементы диаграмм деятельности*

Графическое изображение	Название
Имя состояния действия	Состояние действия (action state)

Таблица 9.1 *(окончание)*

Графическое изображение	Название
Имя состояния под-деятельности	Состояние под-деятельности (subactivity state)
	Простой переход
	Начальное состояние
	Конечное состояние
	Решение или соединение
	Разделение
	Слияние
имя объекта [характеристика состояния]	Объект с характеристикой состояния
	Поток управления между объектами

Глава 10

Диаграмма компонентов (component diagram)

Все рассмотренные ранее диаграммы отражали концептуальные аспекты построения модели системы и относились к логическому уровню представления. Особенность логического представления заключается в том, что оно оперирует понятиями, которые, вообще говоря, не имеют материального воплощения. Другими словами, различные элементы логического представления, такие как классы, ассоциации, состояния и сообщения, не существуют материально. Они лишь отражают наше понимание статической структуры той или иной системы или динамические аспекты ее поведения.

Основное назначение логического представления состоит в анализе структурных и функциональных отношений между элементами модели системы. Однако для создания конкретной физической системы необходимо некоторым образом реализовать все элементы логического представления в конкретные материальные сущности. Для описания таких реальных сущностей предназначен другой аспект модельного представления, а именно — физическое представление модели.

Чтобы пояснить отличие логического от физического представлений, рассмотрим в общих чертах процесс разработки некоторой программной системы. Ее исходным логическим представлением могут служить структурные схемы алгоритмов и процедур, описания интерфейсов и концептуальные схемы баз данных. Однако для реализации этой системы необходимо разработать исходный текст программы на некотором языке программирования (C++, Pascal, Basic/VBA, Java). При этом уже в тексте программы предполагается некоторая организация программного кода, определяемая синтаксисом языка программирования и предполагающая разбиение исходного кода на отдельные модули.

Тем не менее, исходные тексты программы еще не являются окончательной реализацией проекта, хотя и служат фрагментом его физического представления. Очевидно, программная система может считаться реализованной в том случае, когда она будет способна выполнять функции своего целевого предназначения. А это возможно, только если программный код системы будет реализован в форме исполняемых модулей, библиотек классов

и процедур, стандартных графических интерфейсов, файлах баз данных. Именно эти компоненты являются базовыми элементами физического представления системы в нотации языка UML.

Таким образом, полный проект программной системы представляет собой совокупность моделей логического и физического представлений, которые должны быть согласованы между собой. В языке UML для физического представления моделей систем используются так называемые *диаграммы реализации* (implementation diagrams), которые включают в себя две отдельные канонические диаграммы: диаграмму компонентов и диаграмму развертывания. Особенности построения первой из них рассматриваются в этой главе, а второй — в *главе 11*.

Диаграмма компонентов, в отличие от ранее рассмотренных диаграмм, описывает особенности физического представления системы. Диаграмма компонентов позволяет определить архитектуру разрабатываемой системы, установив зависимости между программными компонентами, в роли которых может выступать исходный, бинарный и исполняемый код. Во многих средах разработки модуль или компонент соответствует файлу. Пунктирные стрелки, соединяющие модули, показывают отношения взаимозависимости, аналогичные тем, которые имеют место при компиляции исходных текстов программ. Основными графическими элементами диаграммы компонентов являются компоненты, интерфейсы и зависимости между ними.

Диаграмма компонентов разрабатывается для следующих целей:

- визуализация общей организации структуры исходного кода программной системы;
- спецификация исполнимого варианта программной системы;
- обеспечение многократного использования отдельных фрагментов программного кода;
- представление концептуальной и физической схем баз данных.

В разработке диаграмм компонентов участвуют как системные аналитики и архитекторы, так и программисты. Диаграмма компонентов обеспечивает согласованный переход от логического представления к конкретной реализации проекта в форме программного кода. Одни компоненты могут существовать только на этапе компиляции программного кода, другие — на этапе его исполнения. Диаграмма компонентов отражает общие зависимости между компонентами, рассматривая последние в качестве классификаторов.

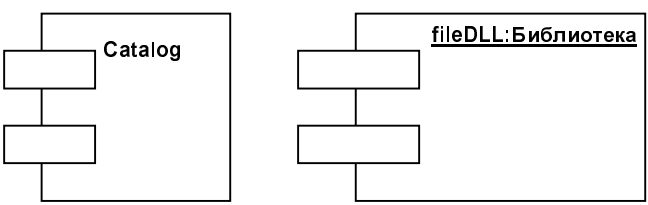
Примечание

Применительно к бизнес-системам программные компоненты следует понимать в более широком контексте, чтобы иметь возможность моделировать бизнес-процессы. В этом случае в качестве компонентов можно рассматривать отдельные организационные подразделения (отделы, службы), должности сотрудников (менеджер, кассир) или документы (счет, заказ, накладная), которые реально существуют в системе.

10.1. Компоненты

Для представления физических сущностей в языке UML применяется специальный термин — *компонент* (component). В метамодели языка UML компонент является потомком классификатора. Он предназначен для представления физической организации ассоциированных с ним элементов модели. Дополнительно компонент может иметь текстовый стереотип и помеченные значения, а некоторые компоненты — собственное графическое представление.

Компонент служит для общего обозначения элементов физического представления модели и может реализовывать некоторый набор интерфейсов. Для графического представления компонента используется специальный символ — прямоугольник со вставленными слева двумя прямоугольниками меньшего размера (рис. 10.1). Внутри большого прямоугольника записывается имя компонента и, возможно, некоторая дополнительная информация. Этот символ является базовым обозначением компонента в языке UML.



а *б*

Рис. 10.1. Графическое изображение компонента в языке UML

Графическое изображение компонента ведет свое происхождение от обозначения модуля программы, применявшегося некоторое время для отображения особенностей инкапсуляции данных и процедур. Так, верхний маленький прямоугольник концептуально ассоциировался с данными, которые реализует этот компонент (ранее он изображался в форме овала). Нижний маленький прямоугольник ассоциировался с операциями или методами, реализуемыми компонентом. В простых случаях имена данных и методов записывались явно в этих маленьких прямоугольниках, однако в языке UML они не указываются.

10.1.1. Имя компонента

Имя компонента подчиняется общим правилам именования элементов модели в языке UML и может состоять из любого числа букв, цифр и некоторых знаков препинания. Отдельный компонент может быть представлен на уровне типа или на уровне экземпляра. Хотя его графическое изображение

в обоих случаях одинаковое, правила записи имени компонента несколько отличаются.

Если компонент представляется на уровне типов, то в качестве его имени записывается только имя типа с заглавной буквы в форме `<Имя типа>`.

Если же компонент представляется на уровне примеров, то в качестве его имени записывается в форме `<имя компонента ':' имя типа>`. При этом вся строка имени подчеркивается.

Так, в первом случае (рис. 10.1, *а*) для компонента уровня типов указывается имя типа, а во втором (рис. 10.1, *б*) для компонента уровня примеров — собственное имя компонента и имя типа.

Примечание

Хотя правила именования объектов в языке UML требуют подчеркивания имени отдельных экземпляров, применительно к компонентам в литературе подчеркивание их имени часто опускают. В этом случае запись имени компонента со строчной буквы характеризует компонент уровня примеров.

В качестве собственных имен компонентов принято использовать имена исполняемых файлов (с расширением EXE), имена динамических библиотек (расширение DLL), имена Web-страниц (расширение HTML), имена текстовых файлов (расширения TXT или DOC) или файлов справки (HLP), имена файлов баз данных (DB) или имена файлов с исходными текстами программ (расширения H, CPP для языка C++, расширение JAVA для языка Java), скрипты (PL, ASP) и другие.

Поскольку конкретная реализация логического представления модели системы зависит от используемого программного инструментария, то и имена компонентов будут определяться особенностями синтаксиса соответствующего языка программирования.

В отдельных случаях к простому имени компонента может быть добавлена информация об имени объемлющего пакета и о конкретной версии реализации данного компонента. Необходимо заметить, что в этом случае номер версии записывается как помеченное значение в фигурных скобках. В других случаях символ компонента может быть разделен на секции, чтобы явно указать имена реализованных в нем интерфейсов. Такое обозначение компонента называется расширенным и рассматривается далее в этой главе.

10.1.2. Виды компонентов

Поскольку компонент как элемент модели может иметь различную физическую реализацию, то иногда его изображают в форме специального графического символа, иллюстрирующего конкретные особенности реализации. Строго говоря, эти дополнительные обозначения не специфицированы в нотации языка UML. Однако, удовлетворяя общим механизмам расширения

языка UML, они упрощают понимание диаграммы компонентов, существенно повышая наглядность графического представления.

Для более наглядного изображения компонентов были предложены и стали общепринятыми следующие графические стереотипы:

- во-первых, стереотипы для компонентов развертывания, которые обеспечивают непосредственное выполнение системой своих функций. Такими компонентами могут быть динамически подключаемые библиотеки с расширением DLL (рис. 10.2, *а*), Web-страницы на языке разметки гипертекста с расширением HTML (рис. 10.2, *б*) и файлы справки с расширением HLP (рис. 10.2, *в*);
- во-вторых, стереотипы для компонентов в форме рабочих продуктов. Как правило — это файлы с исходными текстами программ, как например, с расширениями H или CPP для языка C++ (рис. 10.2, *г*).

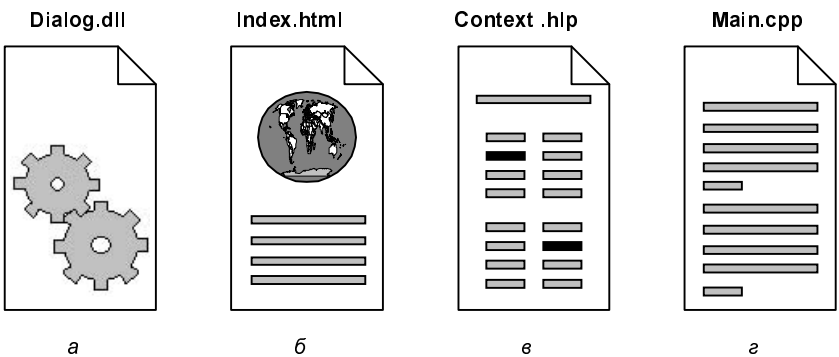


Рис. 10.2. Варианты графического изображения компонентов на диаграмме компонентов

Эти элементы иногда называют *артефактами*, подчеркивая при этом их законченное информационное содержание, зависящее от конкретной технологии реализации соответствующих компонентов. Более того, разработчики могут для этой цели использовать самостоятельные обозначения, поскольку в языке UML нет строгой нотации для графического представления примечаний.

Другой способ спецификации различных видов компонентов — указание текстового стереотипа компонента перед его именем. В языке UML для компонентов определены следующие стереотипы:

- <<file>> (файл) — определяет наиболее общую разновидность компонента, который представляется в виде произвольного физического файла;
- <<executable>> (исполнимый) — определяет разновидность компонента-файла, который является исполнимым файлом и может выполняться на некоторой компьютерной платформе;

- <<document>> (документ) — определяет разновидность компонента-файла, который представляется в форме документа произвольного содержания, не являющегося исполнимым файлом или файлом с исходным текстом программы;
- <<library>> (библиотека) — определяет разновидность компонента-файла, который представляется в форме динамической или статической библиотеки;
- <<source>> (источник) — определяет разновидность компонента-файла, представляющего собой файл с исходным текстом программы, который после компиляции может быть преобразован в исполнимый файл;
- <<table>> (таблица) — определяет разновидность компонента, который представляется в форме таблицы базы данных.

Примечание

Отдельные разработчики предложили собственные графические стереотипы для изображения тех или иных типов компонентов, однако, за исключением стереотипов Дж. Коналлена, они не нашли широкого применения. В свою очередь, некоторые инструментальные CASE-средства также содержат дополнительный набор графических стереотипов для обозначения компонентов. К сожалению, объем книги не позволяет детально остановиться на стереотипах Дж. Коналлена, которые ориентированы для представления диаграмм компонентов Web-приложений. В то же время некоторые из графических стереотипов, реализованные в среде IBM Rational Rose 2002, будут рассмотрены далее в *части III* книги.

10.2. Интерфейсы

Следующим графическим элементом диаграммы компонентов являются интерфейсы. Последние уже упоминались, поэтому здесь будут отмечены те их особенности, которые характерны для их представления на диаграммах компонентов. Напомним, что в общем случае интерфейс графически изображается окружностью, которая соединяется с компонентом отрезком линии без стрелок (рис. 10.3, *а*). При этом имя интерфейса, которое рекомендуется начинать с заглавной буквы "I", записывается рядом с окружностью. Семантически линия означает реализацию интерфейса, а наличие интерфейсов у компонента означает, что данный компонент реализует соответствующий набор интерфейсов.

Другим способом представления интерфейса на диаграмме компонентов является его изображение в виде прямоугольника класса со стереотипом "интерфейс" и возможными секциями атрибутов и операций (рис. 10.3, *б*). Как правило, этот вариант обозначения используется для представления внутренней структуры интерфейса, которая может быть важна для реализации.

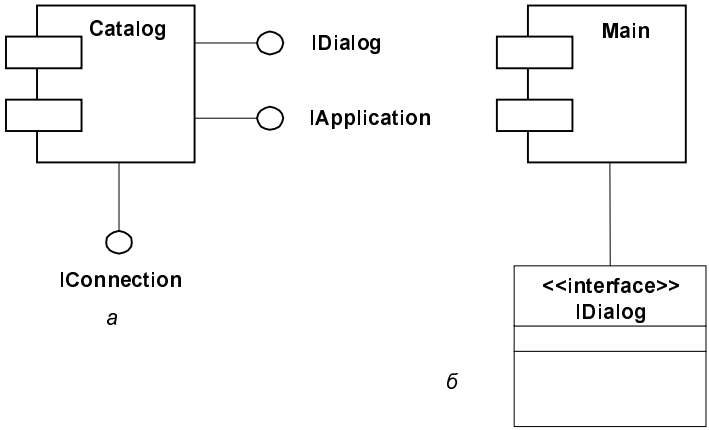


Рис. 10.3. Графическое изображение интерфейсов на диаграмме компонентов

При разработке программных систем интерфейсы обеспечивают не только совместимость различных версий, но и возможность вносить существенные изменения в одни части программы, не изменяя другие ее части. Таким образом, назначение интерфейсов существенно шире, чем спецификация взаимодействия с пользователями системы (актерами).

Характер использования интерфейсов отдельными компонентами может отличаться. Поэтому различают два способа связи интерфейса и компонента. Если компонент реализует некоторый интерфейс, то такой интерфейс называют *экспортируемым*, поскольку он предоставляется в качестве сервиса другим компонентам. Если же компонент использует некоторый интерфейс, который реализуется другим компонентом, то такой интерфейс называется *импортируемым*. Особенность импортируемого интерфейса состоит в том, что на диаграмме компонентов это отношение изображается с помощью зависимости.

10.3. Зависимости

В общем случае отношение зависимости также было рассмотрено ранее (*главы 4* и *5*). Напомним, что отношение зависимости служит для представления факта наличия специальной формы связи между двумя элементами модели, когда изменение одного элемента модели оказывает влияние или приводит к изменению другого элемента модели. Отношение зависимости на диаграмме компонентов изображается пунктирной линией со стрелкой, направленной от клиента (зависимого элемента) к источнику (независимому элементу).

В одних случаях зависимости могут отражать связи отдельных файлов программной системы на этапе компиляции и генерации объектного кода.

В других случаях зависимость может отражать наличие в независимом компоненте описаний классов, которые используются в зависимом компоненте для создания соответствующих объектов. Применительно к диаграмме компонентов зависимости могут связывать компоненты и импортируемые этим компонентом интерфейсы, а также различные виды компонентов между собой.

В этом случае рисуют стрелку от компонента-клиента к импортируемому интерфейсу (рис. 10.4). Наличие такой стрелки означает, что компонент не реализует соответствующий интерфейс, а использует его в процессе своего выполнения. При этом на этой же диаграмме может присутствовать и другой компонент, который реализует этот интерфейс. Отношение реализации интерфейса обозначается на диаграмме компонентов обычной линией без стрелки.

Так, например, изображенный ниже фрагмент диаграммы компонентов (рис. 10.4) представляет информацию о том, что компонент с именем `Main` зависит от импортируемого интерфейса `IDialog`, который, в свою очередь, реализуется компонентом с именем `Library`. При этом для второго компонента этот интерфейс является экспортируемым.

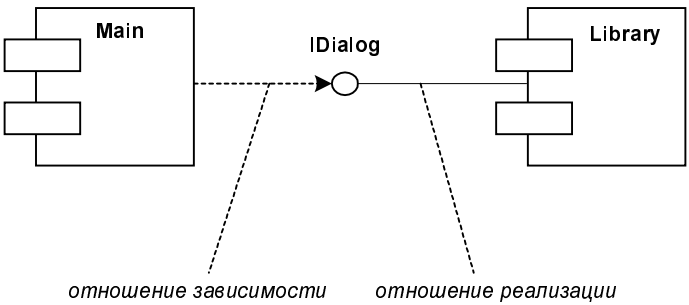


Рис. 10.4. Фрагмент диаграммы компонентов с отношениями зависимости и реализации

Заметим, что изобразить второй компонент с именем `Library` в форме варианта примечания нельзя именно в силу того факта, что этот компонент реализует указанный интерфейс.

Другим случаем отношения зависимости на диаграмме компонентов является отношение программного вызова и компиляции между различными видами компонентов. Для фрагмента диаграммы компонентов (рис. 10.5) наличие подобной зависимости означает, что исполнимый компонент `Main.exe` использует (импортирует) некоторую функциональность компонента `Dialog.dll`, вызывает страницу гипертекста `Index.html` и файл контекстной помощи `Context.hlp`, а исходный текст этого исполнимого компонента хранится в файле `Main.cpp`. При этом характер отдельных видов зависимостей может быть отмечен дополнительно с помощью текстовых стереотипов.

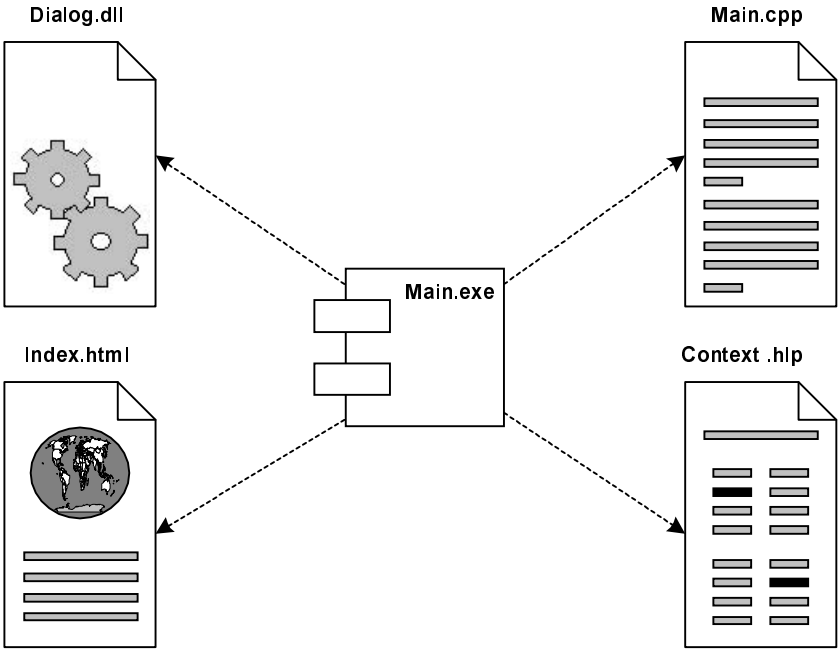


Рис. 10.5. Графическое изображение отношения зависимости между компонентами

Наконец, на диаграмме компонентов могут быть представлены отношения зависимости между компонентами и реализованными в них классами. Эта информация имеет важное значение для обеспечения согласования логического и физического представлений модели системы. Разумеется, изменения в структуре описаний классов могут привести к изменению этой зависимости. Ниже приводится фрагмент зависимости подобного рода, когда некоторый исполнимый компонент `Main.exe` зависит от соответствующих классов (рис. 10.6).

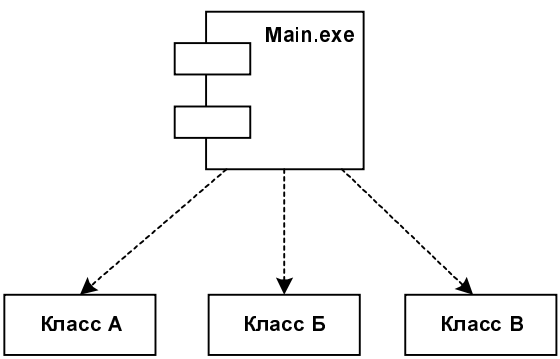


Рис. 10.6. Графическое изображение зависимости между компонентом и классами

Следует заметить, что в данном случае из диаграммы компонентов не следует, что классы реализованы данным компонентом. Если требуется подчеркнуть, что некоторый компонент реализует отдельные классы, то для обозначения компонента используется расширенный символ прямоугольника. При этом прямоугольник компонента делится на две секции горизонтальной линией. Верхняя секция служит для записи имени компонента и, возможно, дополнительной информации, а нижняя секция — для указания реализуемых данным компонентом классов (рис. 10.7).

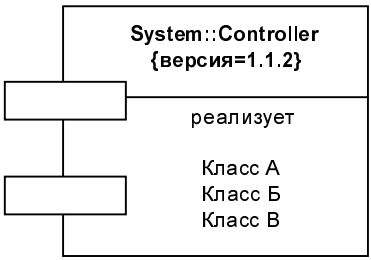


Рис. 10.7. Графическое изображение компонента с информацией о реализуемых им классах

Внутри символа компонента допускается изображать другие элементы графической нотации, такие как классы (компонент уровня типов) или объекты (компонент уровня примеров). В этом случае символ компонента изображается таким образом, чтобы вместить эти дополнительные символы. Так, например, изображенный ниже компонент (рис. 10.8) является экземпляром и реализует три отдельных объекта.

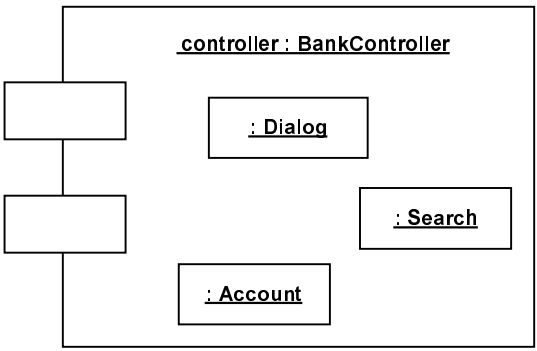


Рис. 10.8. Графическое изображение компонента-экземпляра, реализующего отдельные объекты

Объекты, которые находятся в отдельном компоненте-экземпляре, изображаются вложенными в символ данного компонента. Подобная вложенность

означает, что выполнение компонента влечет выполнение операций соответствующих объектов. Другими словами, существование компонента в течение времени исполнения программы обеспечивает существование и специфицированную функциональность всех вложенных в него объектов. Что касается доступа к этим объектам, то он может быть дополнительно специфицирован с помощью кванторов видимости, подобно видимости пакетов. Содержательный смысл видимости может отличаться для различных видов пакетов.

Так, для компонентов с исходным текстом программы видимость может означать возможность внесения изменений в соответствующие тексты программ с их последующей перекомпиляцией. Для компонентов с исполняемым кодом программы видимость может характеризовать возможность запуска на исполнение соответствующего компонента или вызова реализованных в нем операций или методов.

10.4. Пример построения диаграммы компонентов системы управления банкоматом

Продолжая рассмотрение сквозного примера модели системы управления банкоматом, построим диаграмму компонентов для моделирования физической структуры соответствующей программной системы. В рамках разрабатываемой модели может быть построена единственная диаграмма компонентов, на которой следует изобразить последовательность компиляции отдельных файлов, а также импорт и экспорт отдельных операций. Возможный вариант диаграммы компонентов программной системы управления банкоматом изображен на рис. 10.9.

Данная диаграмма содержит 6 компонентов, 2 из которых являются исполнимыми файлами, один представлен в форме динамической библиотеки, а 3 — в форме текстовых файлов. При этом компоненты с именами `MainATM.cpp` и `Transactions.cpp` содержат исходный код на языке C++, что указывает на особенность их программной реализации. Примечания в форме помеченных значений уточняют реализуемые в этих файлах классы модели. Исполнимый компонент `MainATM` использует операции интерфейса `IAuthorise`, которые, в свою очередь, реализованы в исполнимом компоненте `MainBank`.

Примечание

Изображенная диаграмма компонентов (рис. 10.9) отражает лишь один из возможных вариантов физической реализации рассматриваемой модели. Если в качестве языка программирования использовать не C++, а, например, Java или Pascal, то соответствующим образом изменится и диаграмма компонентов. Выполнить модификацию диаграммы компонентов для этих случаев предлагается читателям самостоятельно в качестве упражнения.

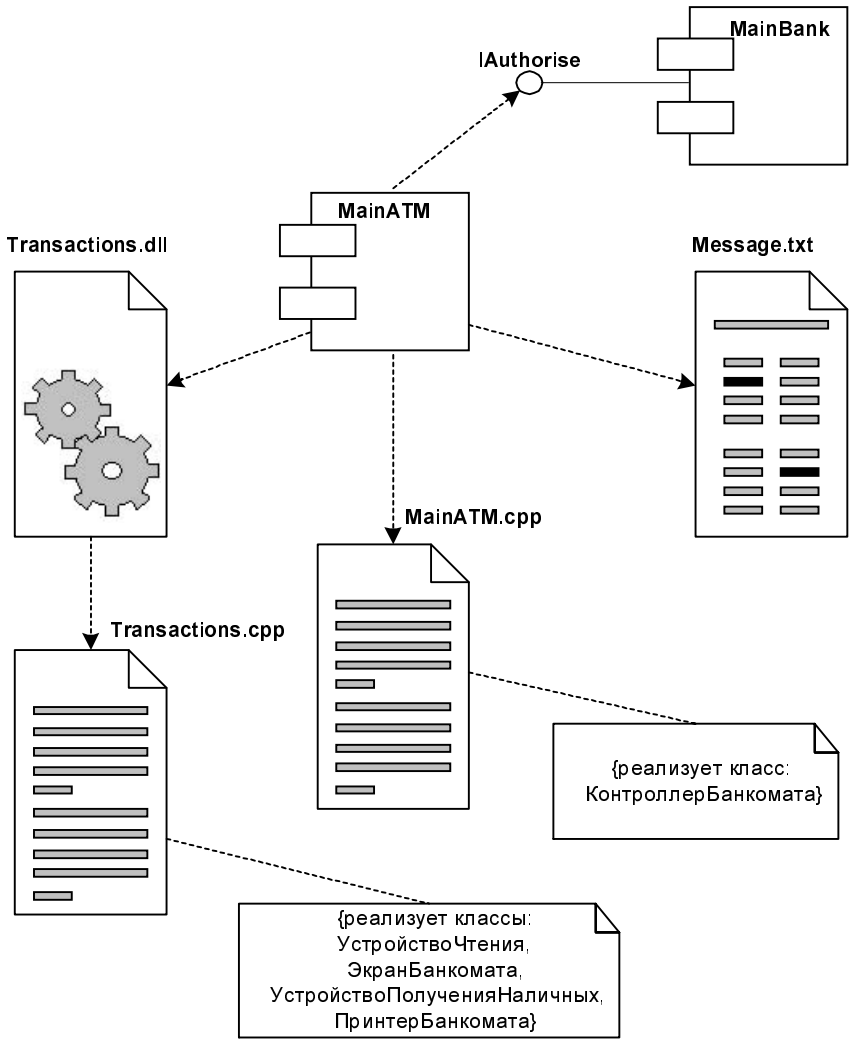


Рис. 10.9. Диаграмма компонентов программной системы управления банкоматом

Следует отметить, что диаграмма компонентов строится только в том случае, если предполагается программная реализация разрабатываемой модели. В противном случае данный тип канонических диаграмм может вообще отсутствовать в модели, что характерно для проектов документирования и реинжиниринга бизнес-процессов.

10.5. Рекомендации по построению диаграммы компонентов

Разработка диаграммы компонентов предполагает использование информации как о логическом представлении модели системы, так и об особенностях ее физической реализации. До начала разработки этой диаграммы необходимо принять решения о выборе вычислительных платформ и операционных систем, на которых предполагается реализовывать систему, а также о выборе конкретных баз данных и языков программирования.

После этого можно приступать к общей структуризации диаграммы компонентов. В первую очередь, необходимо решить, из каких физических частей (файлов) будет состоять программная система. На этом этапе следует обратить внимание на такую реализацию системы, которая обеспечивала бы не только возможность повторного использования кода за счет рациональной декомпозиции компонентов, но и создание объектов только при их необходимости.

Речь идет о том, что общая производительность программной системы существенно зависит от рационального использования ею вычислительных ресурсов. Для этой цели необходимо большую часть описаний классов, их операций и методов вынести в динамические библиотеки, оставив в исполняемых компонентах только самые необходимые для инициализации программы фрагменты программного кода.

После общей структуризации физического представления системы необходимо дополнить модель интерфейсами и схемами базы данных. При разработке интерфейсов следует обращать внимание на согласование (стыковку) различных частей программной системы. Включение в модель схемы базы данных предполагает спецификацию отдельных таблиц и установление информационных связей между ними.

Наконец, завершающий этап построения диаграммы компонентов связан с установлением и нанесением на диаграмму взаимосвязей между компонентами, а также отношений реализации. Эти отношения должны иллюстрировать все важнейшие аспекты физической реализации системы, начиная с особенностей компиляции исходных текстов программ и заканчивая исполнением отдельных частей программы на этапе ее выполнения. Для этой цели можно использовать различные графические стереотипы компонентов.

При разработке диаграммы компонентов следует придерживаться общих принципов создания моделей на языке UML. В частности, в первую очередь необходимо использовать уже имеющиеся в языке UML и общепринятые графические и текстовые стереотипы. Для большинства типовых проектов этого набора элементов может оказаться достаточно для представления компонентов и зависимостей между ними.

Если же проект содержит некоторые физические элементы, описание которых отсутствует в языке UML, то следует воспользоваться механизмом расширения. В частности, можно использовать дополнительные стереотипы для отдельных нетиповых компонентов (например, аплетов и сервлетов) или помеченные значения для уточнения их отдельных характеристик.

Наконец, следует обратить внимание, что диаграмма компонентов, как правило, разрабатывается совместно с диаграммой развертывания, на которой представляется информация о физическом размещении компонентов программной системы по ее отдельным узлам. Особенности построения диаграммы развертывания будут рассмотрены в следующей главе.

В заключение приводится сводка всех рассмотренных графических обозначений, которые могут встретиться при построении диаграмм компонентов (табл. 10.1). Хотя некоторыми авторами в последние годы предложены дополнительные графические стереотипы для изображения отдельных типов компонентов, представленного перечня может оказаться вполне достаточно для выполнения большинства реальных проектов.

Таблица 10.1. *Графические элементы диаграмм компонентов*

Графическое изображение	Название
Имя компонента	Компонент (component)
	Компонент — динамическая библиотека
	Компонент — web-страница

Таблица 10.1 *(окончание)*

Графическое изображение	Название
	Компонент — файл справки (помощи)
	Компонент — файл с исходным текстом программы
	Импортируемый интерфейс (зависимость)
	Экспортируемый интерфейс (реализация)

Глава 11

Диаграмма развертывания (deployment diagram)

Физическое представление программной системы не может быть полным, если отсутствует информация о том, на какой платформе и на каких вычислительных средствах она реализована. Конечно, если разрабатывается простая программа, которая может выполняться локально на компьютере пользователя, не используя никаких распределенных устройств и сетевых ресурсов, то в этом случае нет необходимости в разработке дополнительных диаграмм. Однако при разработке корпоративных приложений ситуация представляется иначе.

Во-первых, сложные программные системы могут реализовываться в сетевом варианте, используя различные вычислительные платформы и технологии доступа к базам данных. Наличие локальной корпоративной сети требует решения целого комплекса дополнительных задач по рациональному размещению компонентов по узлам этой сети, что определяет общую производительность программной системы.

Во-вторых, интеграция программной системы с Интернетом определяет необходимость решения дополнительных вопросов при проектировании системы, таких как обеспечение безопасности, криптозащищенности и устойчивости доступа к информации для корпоративных клиентов. Эти аспекты в немалой степени зависят от реализации проекта в форме физически существующих узлов системы, таких как серверы, рабочие станции, брандмауэры, каналы связи и хранилища данных.

Наконец, технологии доступа и манипулирования данными в рамках общей схемы "клиент-сервер" также требуют размещения больших баз данных в различных сегментах корпоративной сети, их резервного копирования, архивирования, кэширования для обеспечения необходимой производительности системы в целом. Эти аспекты также требуют визуального представления с целью спецификации программных и технологических особенностей реализации распределенных архитектур.

Как было отмечено в *главе 10*, первой из диаграмм физического представления является диаграмма компонентов. Второй формой физического представления программной системы является *диаграмма развертывания*

(синоним — диаграмма размещения). Она применяется для представления общей конфигурации и топологии распределенной программной системы и содержит изображение размещения компонентов по отдельным узлам системы. Кроме того, диаграмма развертывания показывает наличие физических соединений — маршрутов передачи информации между аппаратными устройствами, задействованными в реализации системы.

Диаграмма развертывания предназначена для визуализации элементов и компонентов программы, существующих только на этапе ее исполнения (run-time). При этом представляются только компоненты-экземпляры программы, являющиеся исполнимыми файлами или динамическими библиотеками. Те компоненты, которые не используются на этапе исполнения, на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов. На диаграмме развертывания они не указываются.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними. В отличие от диаграмм логического представления, диаграмма развертывания является единственной для системы в целом, поскольку должна всецело отражать особенности ее реализации. Эта диаграмма, по сути, завершает процесс ООАП для конкретной программной системы, и ее разработка, как правило, является последним этапом спецификации модели.

Цели, преследуемые при разработке диаграммы развертывания, следующие:

- указать размещение исполнимых компонентов программной системы по ее физическим узлам;
- показать физические связи между всеми узлами реализации системы на этапе ее исполнения;
- выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Для обеспечения этих требований диаграмма развертывания разрабатывается совместно системными аналитиками, сетевыми инженерами и системотехниками. Далее рассмотрим отдельные элементы, из которых состоят диаграммы развертывания.

11.1. Узел

Узел (node) представляет собой некоторый физически существующий элемент системы, который может обладать некоторым вычислительным ресурсом. В качестве вычислительного ресурса узла может рассматриваться наличие одного или нескольких процессоров, а также некоторого объема оперативной памяти. В языке UML понятие узла расширено — оно может включать в себя не только вычислительные устройства (процессоры),

но и другие механические или электронные устройства, такие как датчики, принтеры, модемы, цифровые камеры, сканеры и манипуляторы.

Графически узел на диаграмме развертывания изображается в форме трехмерного куба (строго говоря, псевдотрехмерного прямоугольного параллелепипеда). Узел имеет имя, которое указывается внутри этого графического символа. Сами узлы могут представляться как в качестве типов (рис. 11.1, *а*), так и в качестве экземпляров (рис. 11.1, *б*).

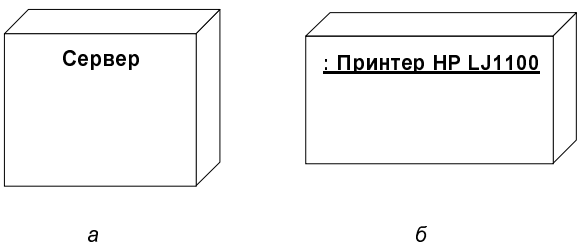


Рис. 11.1. Графическое изображение узла на диаграмме развертывания

В первом случае имя узла записывается в форме: `<Имя типа узла>` без подчеркивания и начинается с заглавной буквы. Во втором — имя узла-экземпляра записывается в виде `<имя узла ':' Имя типа узла>`, а вся запись подчеркивается. Имя типа узла указывает на некоторую разновидность узлов, присутствующих в модели системы.

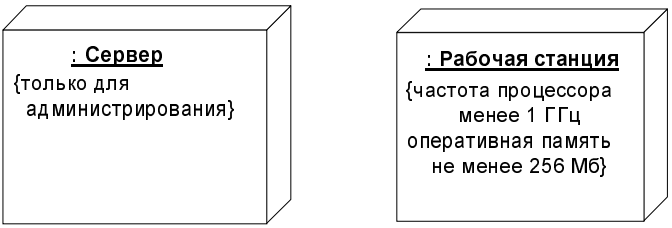
Например, аппаратная часть системы может состоять из нескольких персональных компьютеров, часть из которых выполняют функции сервера и соответствуют отдельным узлам-экземплярам в модели. Однако все эти узлы-экземпляры относятся к одному типу узлов, а именно — узлу с именем типа `Сервер`. Так, на представленном выше рисунке (рис. 11.1, *а*) узел с именем `Сервер` относится к общему типу и никак не конкретизируется. Второй узел (рис. 11.1, *б*) является анонимным узлом-экземпляром конкретной модели принтера.

Так же, как и на диаграмме компонентов, изображения узлов могут расширяться, чтобы включить некоторую дополнительную информацию о спецификации узла. Если дополнительная информация относится к имени узла, то она записывается под этим именем в форме помеченного значения (рис. 11.2).

Если необходимо явно указать компоненты, которые размещаются или выполняются на отдельном узле, то это можно сделать двумя способами. Первый из них позволяет разделить графический символ узла на две секции горизонтальной линией. В верхней секции записывают имя узла, а в нижней секции — размещенные на этом узле компоненты (рис. 11.3, *а*).

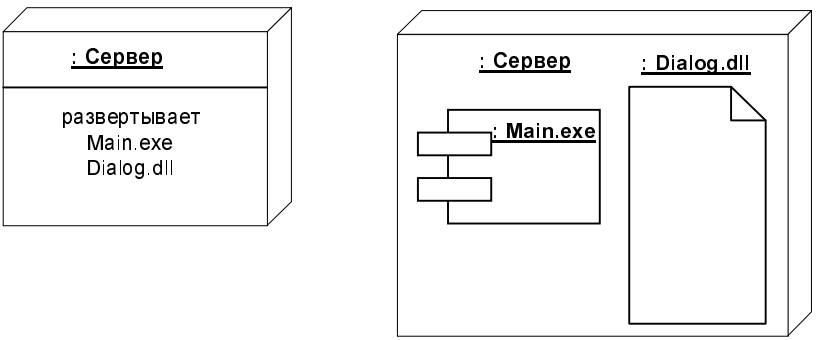
Второй способ разрешает показывать на диаграмме развертывания узлы с вложенными изображениями компонентов (рис. 11.3, *б*). Важно помнить,

что в качестве таких вложенных компонентов могут выступать только исполняемые компоненты и динамические библиотеки.



а *б*

Рис. 11.2. Графическое изображение узла-экземпляра с дополнительной информацией в форме помеченного значения



а *б*

Рис. 11.3. Варианты графического изображения узлов-экземпляров с размещаемыми на них компонентами

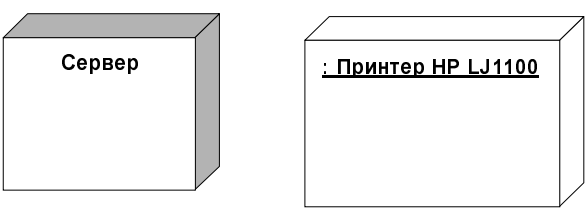
В качестве дополнения к имени узла могут использоваться различные текстовые стереотипы, которые явно специфицируют назначение этого узла. Хотя в языке UML стереотипы для узлов не определены, разработчики предложили следующие текстовые стереотипы: <<processor>> (процессор), <<sensor>> (датчик), <<modem>> (модем), <<net>> (сеть), <<printer>> (принтер) и другие, смысл которых достаточно очевиден.

На диаграммах развертывания допускаются специальные условные обозначения для различных физических устройств, графическое изображение которых проясняет назначение или выполняемые устройством функции. Однако пользоваться этой возможностью следует весьма осторожно, помня, что одно из основных достоинств языка UML следует из его названия — унификация графических элементов визуализации моделей.

Примечание

Хотя возможность включения людей (персонала) в понятие узла не рассматривается в нотации языка UML, подобное расширение понятие узла позволяет создавать средствами языка UML модели самых различных систем, включая бизнес-процессы и технические комплексы. Действительно, для реализации бизнес-логики предприятия может оказаться удобным рассматривать в качестве узлов (ресурсов) системы организационные подразделения, состоящие из персонала. Автоматизация управления техническими комплексами может потребовать рассмотрения в качестве самостоятельного элемента человека-оператора, способного принимать решения в нештатных ситуациях и нести ответственность за возможные последствия этих решений.

Довольно удачным представляются предложенные разработчиками IBM Rational Rose 2002 два специальных графических стереотипа. Первый из них обозначает *ресурсоемкий узел* (processor), под которым понимается узел с наличием процессора и памяти, необходимых для выполнения исполняемых компонентов. Он изображается в форме куба с боковыми гранями, окрашенными в серый цвет (рис. 11.4, *а*).



а *б*

Рис. 11.4. Фрагмент диаграммы развертывания с соединениями между узлами

Второй стереотип в форме обычного куба обозначает *устройство* (device), под которым в среде IBM Rational Rose 2002 понимается узел без процессора и оперативной памяти (рис. 11.4, *б*). На этом типе узлов не могут размещаться исполняемые компоненты программной системы.

Примечание

Кроме известных текстовых и графических стереотипов для узлов диаграммы развертывания самими разработчиками могут быть предложены дополнительные графические стереотипы, которые улучшают наглядность представления диаграмм развертывания. Например, рабочую станцию можно изобразить как в виде ресурсоемкого узла (рис. 11.4, *а*), так и в форме рисунка внешнего вида компьютера. Соответственно, консоль может быть изображена в виде рисунка клавиатуры. В подобных случаях разработчик должен обладать, в дополнение к основным, еще и художественными способностями.

11.2. Соединения и зависимости

На диаграмме развертывания кроме изображения узлов указываются отношения между ними. В качестве отношений выступают физические соединения между узлами, а также зависимости между узлами и компонентами, которые допускается изображать на диаграммах развертывания.

Соединения являются разновидностью ассоциации и изображаются отрезками линий без стрелок. Наличие такой линии указывает на необходимость организации физического канала для обмена информацией между соответствующими узлами. Характер соединения может быть дополнительно специфицирован примечанием, стереотипом, помеченным значением или ограничением. Так, на представленном ниже фрагменте диаграммы развертывания (рис. 11.5) явно определены не только требования к скорости передачи данных в локальной сети с помощью помеченного значения, но и рекомендации по технологии физической реализации соединений в форме примечания.

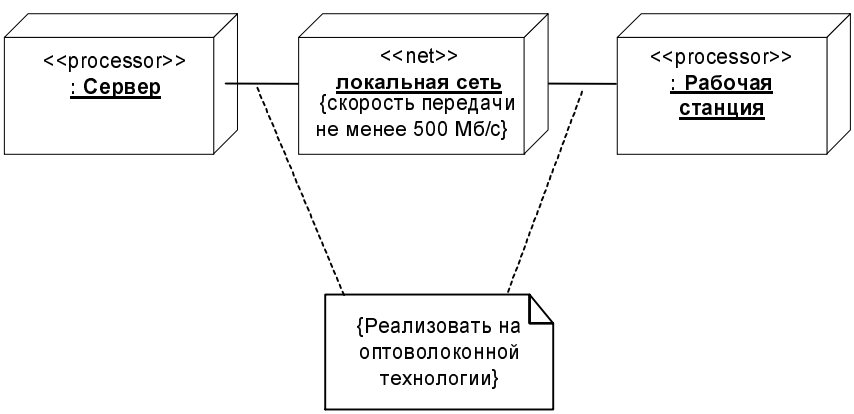


Рис. 11.5. Фрагмент диаграммы развертывания с соединениями между узлами

Кроме соединений на диаграмме развертывания могут присутствовать отношения зависимости между узлом и размещаемыми на нем компонентами. Подобный способ является альтернативой вложенному изображению компонентов внутри символа узла, что не всегда удобно, поскольку делает этот символ излишне объемным. Поэтому при большом количестве развернутых на узле компонентов соответствующую информацию можно представить в форме отношения зависимости (рис. 11.6).

Разработка так называемых встроенных систем реального времени предполагает не только создание программного кода, но и согласование между собой всех аппаратных средств и механических устройств. В качестве примера рассмотрим фрагмент модели управления удаленным механическим средством

типа транспортной платформы. Такая платформа предназначена для перемещения в агрессивных средах, где присутствие человека невозможно в силу целого ряда физических причин.

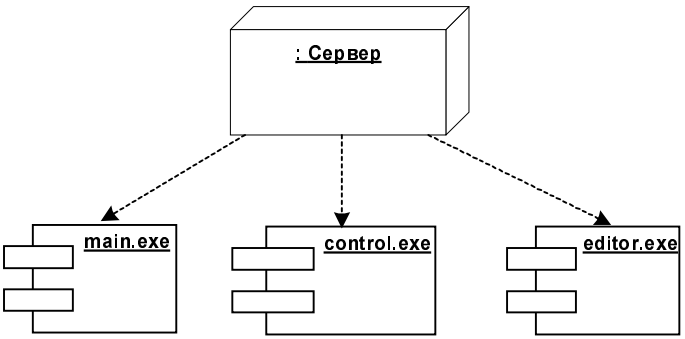


Рис. 11.6. Диаграмма развертывания с отношением зависимости между узлом и развернутыми на нем компонентами

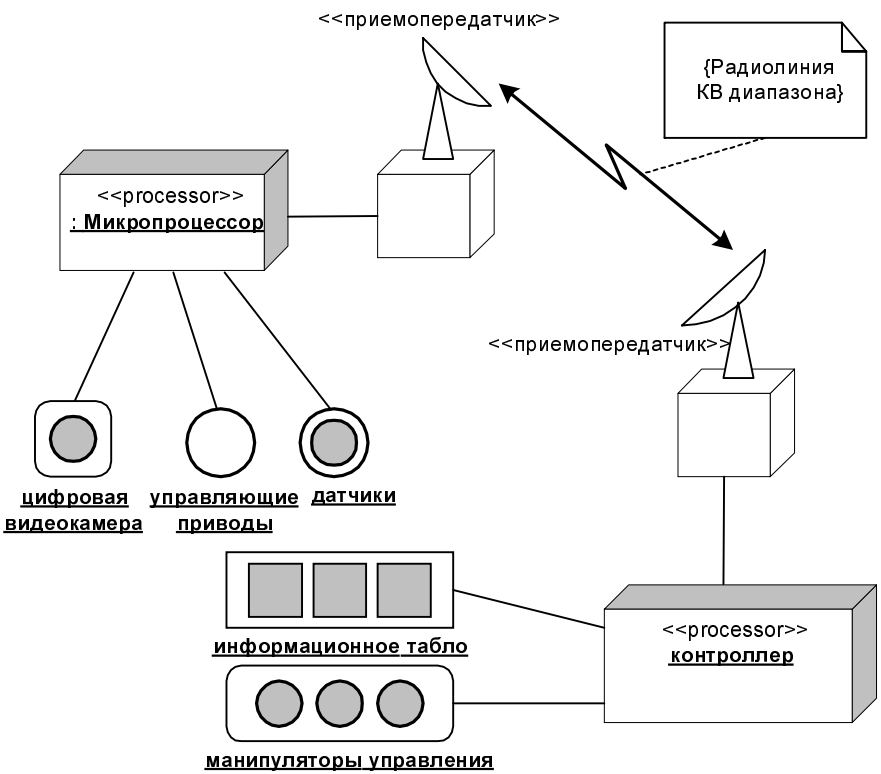


Рис. 11.7. Диаграмма развертывания для модели системы управления транспортной платформой

Транспортная платформа оснащается собственным микропроцессором, цифровой видеокамерой, датчиками температуры и местоположения, а также управляющими приводами для изменения направления и скорости перемещения платформы. Управляющая и телеметрическая информация от платформы по радиолинии передается в центр управления, оснащенный управляющим компьютером, манипуляторами управления и большим информационным табло.

На микропроцессоре платформы развернуты программные компоненты для реализации простейших управляющих воздействий на приводы, что позволяет дискретно изменять направление и скорость перемещения платформы. На компьютере центра управления развернуты программные компоненты анализа телеметрической информации, характеризующей состояние отдельных устройств платформы, а также реализованы алгоритмы управления перемещением платформы в целом.

Вариант физического представления этой транспортной системы показан на следующей диаграмме развертывания (рис. 11.7).

Данная диаграмма содержит самую общую информацию о развертывании рассматриваемой системы и в последующем может быть детализирована при разработке собственно программных компонентов управления. Как видно из рисунка, при разработке этой диаграммы развертывания был использован дополнительный стереотип <<приемопередатчик>>, который отсутствует в описании языка UML, и специальные графические изображения (стереотипы) для отдельных аппаратных и механических устройств.

11.3. Пример построения диаграммы развертывания системы управления банкоматом

Продолжая рассмотрение сквозного примера модели системы управления банкоматом, построим диаграмму развертывания для моделирования физической структуры соответствующей программной системы. В рамках разрабатываемой модели может быть построена единственная диаграмма развертывания, на которой следует изобразить отдельные узлы и физические каналы коммуникации между ними. Возможный вариант диаграммы развертывания уровня примеров для программной системы управления банкоматом изображен на рис. 11.8.

Данная диаграмма содержит 7 компонентов, 3 из которых изображены в форме ресурсоемких узлов, а 4 — в форме устройств. При этом на сервере банка и удаленном терминале размещены отдельные компоненты модели. Хотя графическое изображение этих узлов делает излишним указание текстовых стереотипов, однако они оставлены для большей наглядности.

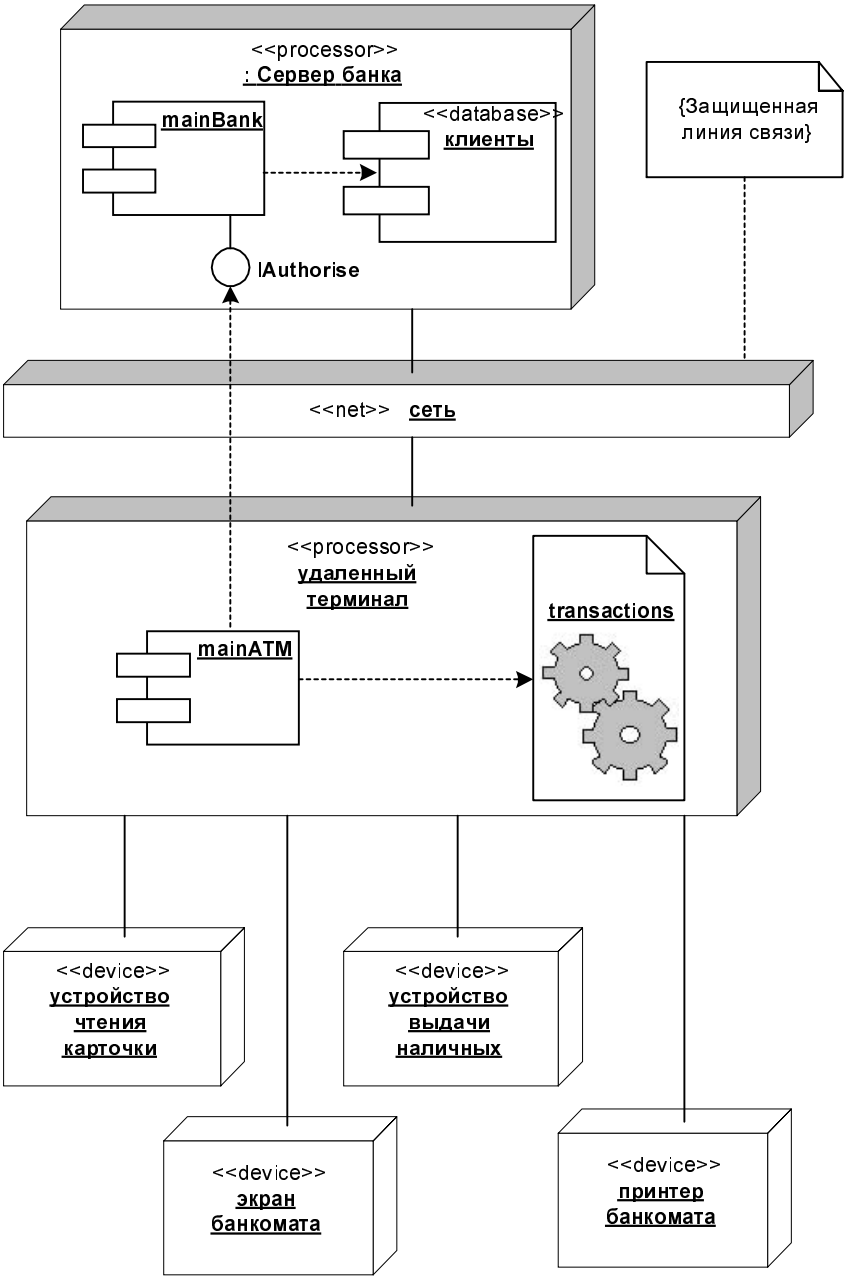


Рис. 11.8. Диаграмма развертывания системы управления банкоматом

Невольный вопрос может возникнуть по поводу изображения сети в качестве ресурсоемкого узла. Однако этот факт можно интерпретировать как желание

разработчика акцентировать внимание на необходимости обеспечения повышенных мер безопасности при передаче информации по этой сети. Это обстоятельство дополнительно отмечено в соответствующем примечании. Примечание указывает на необходимость использования защищенной линии связи для обмена данными в этой системе. Другой вариант записи этой информации заключается в изменении стереотипа узла сети на <<private net>>.

Как уже отмечалось ранее, диаграммы развертывания могут иметь более сложную структуру, включающую вложенные компоненты, интерфейсы и другие аппаратные устройства. На рассматриваемой диаграмме развертывания (рис. 11.8) явно указана зависимость компонента `mainATM` на удаленном терминале от интерфейса `IAuthorise`, который реализован компонентом `mainBank`, который, в свою очередь, размещен на анонимном узле-экземпляре `Сервер банка`. Компонент `mainBank` зависит от компонента базы данных с именем `клиенты`, который развернут на этом же узле.

Примечание

Изображенная диаграмма компонентов (см. рис. 11.8) отражает лишь один из возможных вариантов физической реализации рассматриваемой модели. Не вызывает сомнения тот факт, что на разработку конкретной диаграммы развертывания оказывают влияние целый ряд технологических, экономических, финансовых, организационных и других факторов. При этом в контексте нотации языка UML именно данный тип канонических диаграмм оказывается наиболее удобным для представления соответствующих ограничений, представляющих отмеченные факторы и технологические требования.

Следует отметить, что диаграмма развертывания строится только в том случае, если не только предполагается программная реализация разрабатываемой модели, но и распределенность физического размещения ее компонентов. В противном случае данный тип канонических диаграмм может отсутствовать в модели, что характерно как для проектов документирования и реинжиниринга бизнес-процессов, так и программ, выполняющихся на одном компьютере.

11.4. Рекомендации по построению диаграммы развертывания

Разработка диаграммы развертывания начинается с идентификации всех аппаратных, механических и других типов устройств, которые необходимы для выполнения системой всех своих функций. В первую очередь специфицируются вычислительные узлы системы, обладающие процессором и оперативной памятью. При этом используются имеющиеся в языке UML стереотипы, а в случае их отсутствия разработчики могут определить новые стереотипы. Отдельные требования к составу аппаратных средств могут быть заданы в форме ограничений и помеченных значений.

Дальнейшая детализация диаграммы развертывания связана с размещением всех исполняемых компонентов диаграммы по узлам системы. В модели должна быть исключена ситуация, когда отдельные исполняемые компоненты оказались не размещенными на узлах. Для чего может потребоваться внести в модель дополнительные узлы, содержащие процессор и оперативную память.

Как уже отмечалось, при разработке простых программ, которые исполняются локально на одном компьютере, необходимость в диаграмме развертывания может вообще отсутствовать. В более сложных ситуациях диаграмма развертывания строится для следующих приложений.

- Моделирование программных систем, реализующих технологию доступа к данным "клиент-сервер". Для подобных систем характерно четкое разделение полномочий и, соответственно, компонентов между клиентскими рабочими станциями и сервером базы данных. Возможность реализации "тонких" клиентов на простых терминалах или организация доступа к хранилищам данных приводит к необходимости уточнения не только топологии системы, но и ее компонентного состава.
- Моделирование неоднородных распределенных архитектур. Речь идет о корпоративных интрасетях, насчитывающих сотни компьютеров и других периферийных устройств, функционирующих на различных платформах и под различными операционными системами. При этом отдельные узлы такой системы могут быть удалены друг от друга на сотни километров (филиалы компаний). В этом случае диаграмма развертывания становится важным инструментом визуализации общей топологии системы и контроля миграции отдельных компонентов между узлами.
- Наконец, уже упоминавшиеся ранее, системы реального времени со встроенными микропроцессорами, которые могут функционировать автономно. Такие системы могут содержать самые разнообразные дополнительные устройства, обеспечивающие автономность их функционирования при решении сложных целевых задач. Для подобных систем диаграмма развертывания позволяет визуализировать состав всех устройств и их взаимосвязи в системе.

Как правило, разработка диаграммы развертывания осуществляется на завершающем этапе ООАП, что характеризует окончание фазы проектирования физического представления. С другой стороны, диаграмма развертывания может строиться для анализа существующей системы с целью ее последующего анализа и модификации. При этом анализ предполагает разработку этой диаграммы на его начальных этапах, что характеризует общее направление анализа от физического представления к логическому.

При моделировании бизнес-процессов диаграмма развертывания, кроме компьютеров корпоративной сети, может содержать в качестве узлов раз-

личные средства оргтехники (факсимильные устройства, многоканальные телефонные станции, множительные аппараты, экраны для презентаций и другие). При этом каждое из подобных устройств может функционировать как автономно, так и в составе корпоративной сети.

Если необходимо включить в модель ресурсы Интернета, то Интернет часто обозначается в форме "облачка" с соответствующим именем. Строго говоря, подобное обозначение не специфицировано в языке UML, однако оно является общепринятым при разработке моделей Web-приложений.

Наконец, следует отметить одно немаловажное обстоятельство, характерное для разработки всех канонических диаграмм. Хотя в языке UML определена графическая нотация для всех элементов канонических диаграмм, способы графического изображения отдельных инструментальных средств имеют свои специфические особенности. Применение того или иного CASE-средства накладывает определенные ограничения на визуализацию моделей программных систем. Речь идет о том, что некоторые элементы языка UML могут вообще отсутствовать в CASE-средствах. Выход из подобной ситуации может быть связан либо с выбором другого инструментария, поддерживающего последние версии языка UML, либо упрощении модели на основе ее типизации.

В *главе 12* некоторые из этих аспектов будут рассмотрены более подробно на примере CASE-средства IBM Rational Rose 2002.

В заключение приводится сводка всех рассмотренных графических обозначений, которые могут использоваться при построении диаграмм развертывания (табл. 11.1). То обстоятельство, что этот перечень невелик по объему, ярко демонстрирует результаты действительной унификации элементов модели в нотации языка UML.

Таблица 11.1. *Графические элементы диаграмм развертывания*

Графическое изображение	Название
Имя узла	Узел (node)
Имя процессора	Ресурсоемкий узел (процессор)

Таблица 11.1 *(окончание)*

Графическое изображение	Название
Имя устройства	Нересурсоемкий узел (устройство)
	Физическое соединение узлов
	Зависимость (dependency)

Часть III

Анализ и проектирование с использованием нотации языка UML и CASE-средства IBM Rational Rose 2002

Глава 12. Общая характеристика инструментария IBM Rational Rose 2002

Глава 13. Начало работы над проектом в среде IBM Rational Rose 2002

Глава 14. Завершение работы над проектом в среде IBM Rational Rose 2002

Появление на рынке программных продуктов первых CASE-средств (Computer Aided Software Engineering) ознаменовало новый этап развития программной инженерии, характерными особенностями которого являются существенное сокращение сроков реализации программных проектов, поддержка методологии групповой разработки и ориентация на визуальные средства специфицирования компонентов программного обеспечения.

Первоначальной областью применения CASE-средств были приложения баз данных, особенно те из них, которые требовали серьезных усилий при разработке своих концептуальных схем. Реализация возможности автоматической генерации программного кода на основе предварительно разработанной концептуальной схемы оказалась настолько конструктивной, что стимулировала появление более двух десятков CASE-средств различных фирм.

Как уже отмечалось в *части I* книги, начальный этап развития CASE-технологий характеризовался тем, что разные фирмы предлагали свои собственные средства визуального представления концептуальных схем. Зачастую выбор разработчиками того или иного CASE-средства определялся простотой нотации поддерживаемого средством языка представления схем и диаграмм. Появление первых стандартов в этой области лишь на какое-то время стабилизировало ситуацию. Однако острейшая конкуренция среди фирм-производителей программного обеспечения требовала от CASE-средств реализации объектно-ориентированной технологии разработки программ и поддержки широкого диапазона языков программирования и конкретных баз данных.

В настоящее время рынок CASE-средств пополнился десятками новых инструментов, возможности которых еще недавно считались маловероятными. В первую очередь, следует отметить появление средства ModelMaker, которое поставляется вместе со средой Borland Delphi 7 и позволяет генерировать код на языке Delphi Pascal, с возможностью последующей отладки и получения исполняемых модулей в среде Borland Delphi 7. Не менее впечатляющей возможностью ModelMaker является поддержка паттернов проектирования в контексте нотации языка UML.

Следует упомянуть также реализацию аналогичных возможностей в среде MS Visual Studio.NET, которая поддерживает разработку диаграмм языка UML и последующую генерацию программного кода в нотации соответствующих языков программирования, включая новый язык C#.

Однако, по универсальности платформ реализации, полноте языков программирования и схем баз данных, продолжает лидировать средство IBM Rational Rose 2002. Именно по этой причине данное средство выбрано в качестве базового для иллюстрации возможностей инструментальной поддержки языка UML и процесса разработки визуальных моделей в соответствующей нотации.

Глава 12

Общая характеристика инструментария IBM Rational Rose 2002

Среди всех фирм-производителей CASE-средств именно компания IBM Rational Software Corp. (до августа 2003 г. — Rational Software Corp.) одна из первых осознала стратегическую перспективность развития объектно-ориентированных технологий анализа и проектирования программных систем. Эта компания выступила инициатором унификации языка визуального моделирования в рамках консорциума OMG, что, в конечном итоге, привело к появлению первых версий языка UML. И эта же компания первой разработала инструментальное объектно-ориентированное CASE-средство, в котором был реализован язык UML как базовая нотация визуального моделирования.

Среди причин, сдерживающих применение CASE-средств и определяющих контраст их популярности среди западных и отечественных разработчиков программ, следует отметить, в первую очередь, масштабность проектов и различие в технологиях создания программ. С одной стороны, необходимость автоматизации анализа и проектирования программных систем на базе CASE-технологии начинает осознаваться только тогда, когда выполняемый проект является достаточно сложным и масштабным. В противном случае для написания программ вполне достаточно обычных инструментов разработчика.

С другой стороны, реализация масштабных проектов под силу группе разработчиков высокой квалификации, в которую, кроме программистов, должны входить системные аналитики, бизнес-аналитики, системотехники, тестировщики и другие категории специалистов. При этом непременным условием успешной работы является соответствующая квалификация специалистов, в том числе и знание ими базовой нотации языка UML, а обеспечение групповой работы над проектом требует дополнительных средств для обеспечения совместимости его составных частей.

12.1. Общая характеристика CASE-средства IBM Rational Rose 2002

CASE-средство IBM Rational Rose со времени своего появления претерпело серьезную эволюцию и, в настоящее время, представляет собой современный интегрированный инструментарий для анализа, моделирования и разработки программных систем. Именно в IBM Rational Rose язык UML стал базовой технологией визуализации и разработки программных систем, что определило популярность и стратегическую перспективность этого средства.

В рамках IBM Rational Rose существуют различные варианты этого средства, отличающиеся между собой диапазоном реализованных возможностей. Базовым средством в настоящее время является IBM Rational Rose Enterprise Edition, которое обладает наиболее полными возможностями.

Последней версией этого CASE-средства на момент написания книги является IBM Rational Rose 2002 (release 2002.05.00), возможности которой аккумулируют практически все современные достижения в области информационных технологий:

- интеграция с MS Visual Studio 6/.NET, что включает в себя поддержку на уровне прямой и обратной генерации кодов и диаграмм VB 6, Visual C++ 6, Visual J++ 6 (ATL — Microsoft Active Template Library, WebClasses, DHTML, Data Connections);
- непосредственная работа (инжиниринг и реинжиниринг) с исполняемыми модулями и библиотеками;
- поддержка технологий MTS (Microsoft Transaction Server) и ADO (ActiveX Data Objects) на уровне шаблонов и исходного кода, а также элементов технологии COM+ (DCOM);
- полная поддержка CORBA, включая реализацию технологии компонентной разработки приложений CBD (Component-Based Development), языка определения интерфейса IDL (Interface Definition Language) и языка определения данных DDL (Data Definition Language);
- полная поддержка среды разработки Java-приложений, включая прямую и обратную генерацию классов Java формата JAR, а также работу с файлами формата CAB и ZIP.

Уже этого перечня основных особенностей может оказаться достаточно, чтобы сделать вывод о достижении совершенно нового уровня реализации CASE-технологий, когда само инструментальное средство становится не только рабочим инструментом, но и своеобразной базой данных для практически всех современных объектных стандартов и компонентных интерфейсов.

Примечание

Конечно, рассмотреть в одной главе возможности такого средства, как IBM Rational Rose 2002, просто невозможно, и автор не ставил перед собой такую задачу. Цель нашего знакомства с этим инструментарием — осветить основные особенности реализации языка UML на уровне разработки отдельных диаграмм. Поэтому далее описываются лишь основные правила и рекомендации, необходимые для разработки визуальных моделей в форме канонических диаграмм языка UML, реализованные в среде IBM Rational Rose 2002.

12.2. Особенности рабочего интерфейса IBM Rational Rose 2002

В CASE-средстве IBM Rational Rose 2002 реализованы общепринятые стандарты для рабочего интерфейса программы, подобно известным средам визуального программирования. После установки IBM Rational Rose 2002 на компьютер пользователя, что практически не вызывает трудностей, запуск этого средства в операционных системах семейств Microsoft (MS) Windows 9x/NT приводит к появлению на экране соответствующего рабочего интерфейса программы (рис. 12.1).

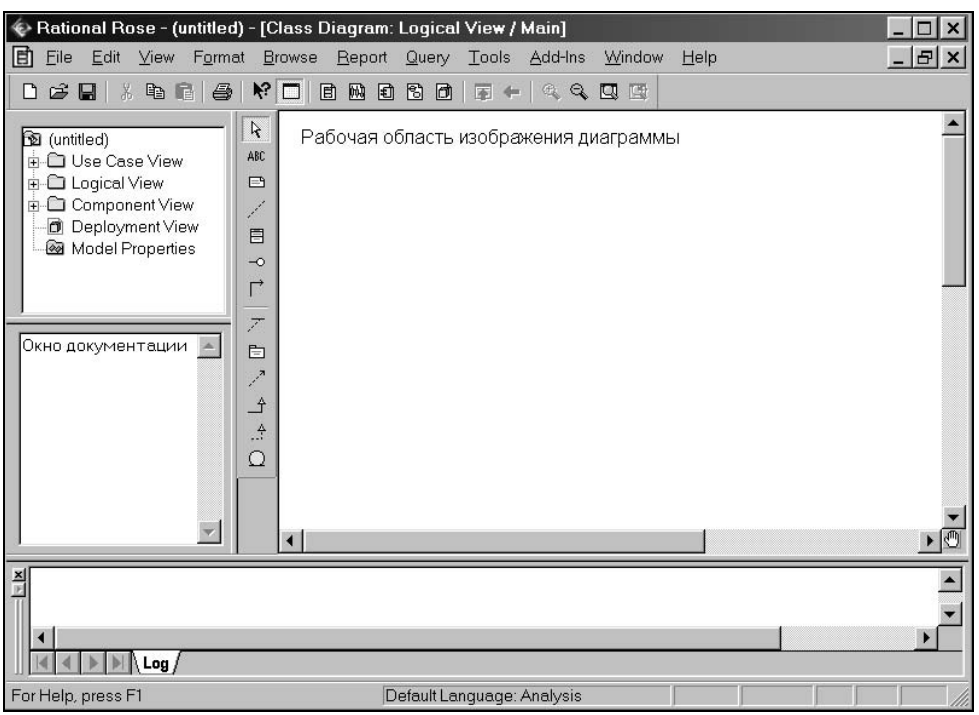


Рис. 12.1. Общий вид рабочего интерфейса средства Rational Rose 2002

Рабочий интерфейс состоит из различных элементов, основными из которых являются:

- главное меню;
- стандартная панель инструментов;
- окно браузера;
- специальная панель инструментов;
- рабочая область изображения диаграммы (окно диаграммы);
- окно документации;
- окно журнала.

Рассмотрим кратко назначение и основные функции каждого из этих элементов.

12.2.1. Главное меню

Главное меню средства IBM Rational Rose 2002 выполнено в общепринятом стандарте и выглядит следующим образом (рис. 12.2).

Рис. 12.2. Внешний вид главного меню программы

Отдельные пункты меню объединяют сходные операции, относящиеся ко всему проекту в целом. Некоторые из пунктов меню содержат хорошо знакомые функции (открытие проекта, вывод на печать диаграмм, копирование в буфер и вставка из буфера различных элементов диаграмм). Другие настолько специфичны, что могут потребовать дополнительных усилий на изучение (опции генерации программного кода, проверка согласованности моделей, подключение дополнительных модулей).

12.2.2. Стандартная панель инструментов

Стандартная панель инструментов располагается ниже строки главного меню и выглядит следующим образом (рис. 12.3). Некоторые из инструментов недоступны для нового проекта, который пока не имеет никаких элементов. Стандартная панель инструментов обеспечивает быстрый доступ к тем командам меню, которые выполняются разработчиками наиболее часто.

Рис. 12.3. Внешний вид стандартной панели инструментов

Пользователь может настроить внешний вид этой панели по своему усмотрению. Для этого необходимо выполнить операцию главного меню **Tools → Options** (Инструменты → Параметры), открыть вкладку **Toolbars** (Панели инструментов) появившегося диалогового окна и нажать кнопку **Standard** (Стандартная). В дополнительно открытом окне можно переносить требуемые кнопки из левого списка в правый список, а ненужные кнопки — из правого списка в левый. Данным способом можно показать или скрыть различные кнопки инструментов, а также изменить их размер. Назначение отдельных кнопок стандартной панели инструментов приводится далее (см. табл. 12.1).

12.2.3. Окно браузера проекта

Окно браузера проекта по умолчанию располагается в левой части рабочего интерфейса под стандартной панелью инструментов и имеет следующий вид (рис. 12.4).

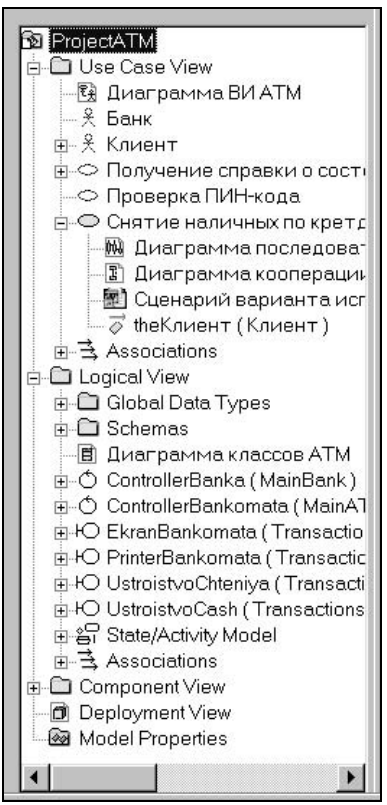


Рис. 12.4. Внешний вид браузера с иерархическим представлением структуры проекта

Браузер проекта организует представления модели в виде иерархической структуры, которая упрощает навигацию и позволяет отыскать любой элемент модели в проекте. При этом любой элемент, который разработчик добавляет в модель, сразу отображается в окне браузера. Соответственно, выбрав элемент в окне браузера, мы можем его визуализировать в окне диаграммы или изменить его спецификацию.

Браузер проекта позволяет также организовывать элементы модели в пакеты и перемещать элементы между различными представлениями модели. При желании окно браузера можно расположить в другом месте рабочего интерфейса либо скрыть вовсе, используя для этого пункт меню **View** (Вид). Можно также изменить размеры браузера, переместив мышью границу его внешней рамки.

12.2.4. Специальная панель инструментов

Специальная панель инструментов располагается между окном браузера и окном диаграммы в средней части рабочего интерфейса. По умолчанию предлагается панель инструментов для построения диаграммы классов модели (рис. 12.5).

Рис. 12.5. Внешний вид специальной панели инструментов для диаграммы классов

Расположение специальной панели инструментов можно изменять, переместив рамку панели в нужное место. Можно настраивать и состав панели, добавляя или удаляя отдельные кнопки, соответствующие тем или иным инструментам. Назначения кнопок можно узнать из всплывающих подсказок, появляющихся после задержки указателя мыши над соответствующей кнопкой.

Примечание

Следует заметить, что внешний вид специальной панели инструментов определяется не только выбором вида разрабатываемой диаграммы, но и выбором графической нотации для изображения самих элементов этих диаграмм. В IBM Rational Rose 2002 реализованы три таких нотации: UML, OMT и Booch. Речь идет о том, что одна и та же диаграмма может быть представлена различным образом, для этого достаточно выбрать желаемое представление через пункт меню **View** (Вид). При этом никаких дополнительных действий выполнять не требуется — диаграмма преобразуется в выбранную нотацию автоматически. Однако, рассматривая IBM Rational Rose 2002 в контексте только языка UML, мы оставим без внимания особенности двух других нотаций, которые отражают эволюционный аспект этого средства.

Более подробно состав и назначение отдельных кнопок специальных панелей инструментов будут рассмотрены далее в *главах 13* и *14* при построении соответствующих типов канонических диаграмм.

12.2.5. Окно диаграммы

Окно диаграммы является основной рабочей областью ее интерфейса, в которой визуализируются различные представления модели проекта. По умолчанию окно диаграммы располагается в правой части рабочего интерфейса, однако его расположение и размеры также можно изменить. При разработке нового проекта, если не был использован мастер проектов, окно диаграммы представляет собой чистую область, не содержащую никаких элементов модели (рис. 12.6).

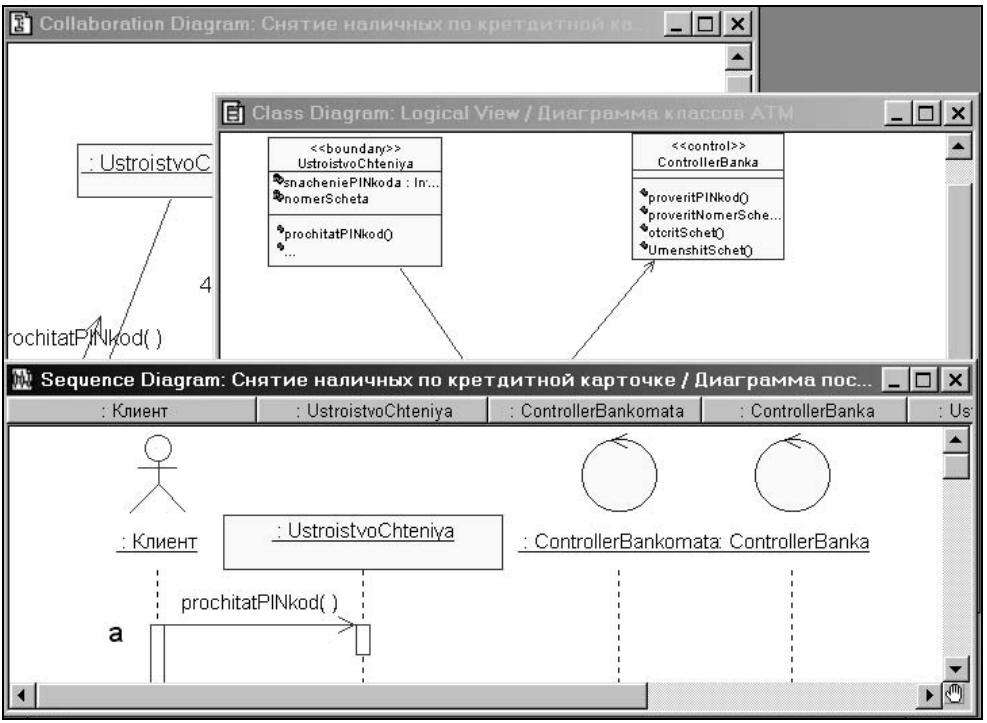


Рис. 12.6. Внешний вид окна диаграмм с различными видами представлений модели

Название диаграммы, которая располагается в данном окне, указывается в строке заголовка программы или, если окно не развернуто во весь экран, в строке заголовка окна диаграммы. Одновременно в окне диаграммы могут присутствовать несколько диаграмм, однако активной может быть только одна из них. Например, на рис. 12.6 активной является диаграмма последо-

вательности, хотя имеются и другие диаграммы. Переключение между диаграммами можно осуществить выбором нужного представления на стандартной панели инструментов либо через пункт меню **Window** (Окно). При активизации отдельного вида диаграммы изменяется внешний вид специальной панели инструментов, которая настраивается под конкретный вид диаграммы.

12.2.6. Окно документации

Окно документации, по умолчанию, присутствует на экране после загрузки программы. Если по какой-то причине оно отсутствует, то его можно активизировать через пункт меню **View → Documentation** (Вид → Документация), после чего окно документации появится ниже окна браузера (рис. 12.7).

Рис. 12.7. Внешний вид окна документации

Окно документации, как следует из его названия, предназначено для документирования элементов представления модели. В него можно записывать самую различную информацию, и что важно — на русском языке. Эта информация в последующем преобразуется в комментарии и никак не влияет на логику выполнения программного кода.

В окне документации активизируется та информация, которая относится к отдельному выделенному элементу диаграммы. При этом выделить элемент можно либо в окне браузера, либо в окне диаграммы. При добавлении нового элемента на диаграмму (например, класса) документация к нему является пустой (No documentation). В последующем разработчик самостоятельно вносит необходимую пояснительную информацию, которая запоминается и может быть изменена в ходе работы над проектом. Так же, как и для других окон рабочего интерфейса IBM Rational Rose 2002, разработчик по своему усмотрению может изменять размеры и положение окна документации.

12.2.7. Окно журнала

Окно журнала (Log) предназначено для автоматической записи различной служебной информации, образующейся в ходе работы с программой. В журнале фиксируется время и характер выполняемых разработчиком действий,

таких как обновление модели, настройка меню и панелей инструментов, а также сообщений об ошибках, возникающих при генерации программного кода.

Окно журнала всегда присутствует на рабочем интерфейсе в области окна диаграммы (рис. 12.8). Однако оно может быть закрыто другими окнами с диаграммами или может быть свернутым.

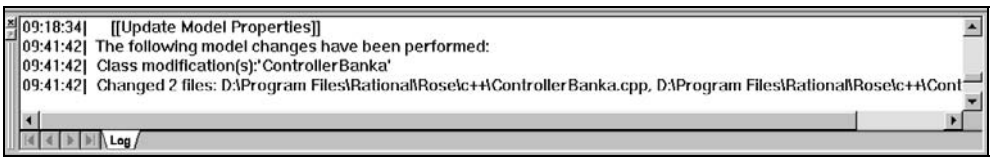


Рис. 12.8. Внешний вид окна журнала

Активизировать окно журнала можно через меню **Window** → **Log** (Окно → Журнал). В этом случае окно изображается поверх других, в правой области рабочего интерфейса. Полностью удалить это окно нельзя, его можно только минимизировать.

12.3. Назначение операций главного меню

Главное меню позволяет пользователю вызывать другие графические средства работы с системой IBM Rational Rose 2002, загружать и сохранять информацию в файлах, изменять внешний вид элементов графического интерфейса, вызывать справочную информацию и т. д. Рассмотрим назначение отдельных пунктов главного меню.

- Пункт меню **File** (Файл) главного меню содержит следующие операции (рис. 12.9).
  - **New** — создает новую модель IBM Rational Rose 2002. При этом новая модель по умолчанию имеет имя untitled.
  - **Open** — вызывает стандартное диалоговое окно открытия внешнего файла с диска. Открыть можно либо файл модели (файл с расширением mdl), либо файл подмодели (файл с расширением ptl).
  - **Save** — позволяет сохранить разрабатываемую модель в файле на диске.
  - **Save As** — позволяет сохранить разрабатываемую модель под другим именем в файле на диске. При этом вызывается стандартное диалоговое окно сохранения файла на диске с предложением задать имя соответствующего файла модели или подмодели.
  - **Save Log As** — позволяет сохранить содержание журнала в файле на диске с именем error.log. При этом вызывается стандартное диалоговое окно сохранения файла на диске с предложением изменить предлагаемое по умолчанию имя соответствующего файла.

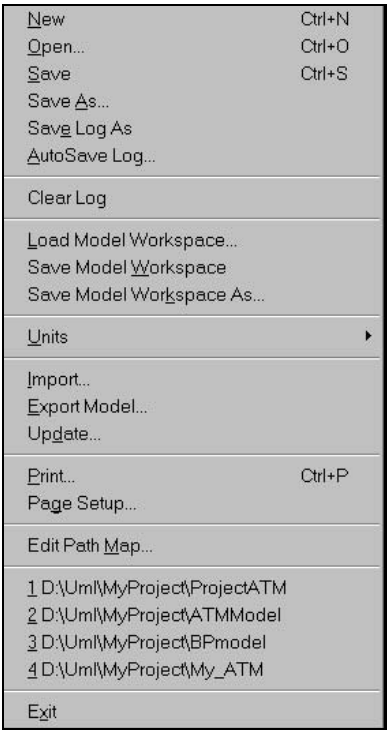


Рис. 12.9. Операции пункта меню **File** (Файл) главного меню

- **AutoSave Log** — позволяет автоматически сохранять содержание журнала в файле на диске с именем error.log. При первом выполнении этого пункта меню также вызывается стандартное диалоговое окно сохранения файла на диске с предложением изменить используемый по умолчанию путь и имя файла.
- **Clear Log** — очищает журнал.
- **Load Model Workspace** — позволяет загрузить рабочую область из файла на диске. Вызывает стандартное диалоговое окно открытия файла, при этом можно открыть только файл с расширением wsp.
- **Save Model Workspace** — позволяет сохранить рабочую область модели в файле на диске. При первом выполнении этого пункта меню вызывается стандартное диалоговое окно сохранения файла с расширением wsp.
- **Save Model Workspace As** — позволяет сохранить рабочую область модели в файле на диске. Вызывается стандартное диалоговое окно сохранения файла с предложением изменить предлагаемое по умолчанию имя соответствующего файла.

- **Units** — содержит вложенные подпункты меню для обеспечения контроля версий разрабатываемой модели.
- **Import** — позволяет импортировать информацию из файлов различных форматов, включая файлы моделей, подмоделей, категорий и подсистем.
- **Export Model** — позволяет экспортировать информацию о модели в файл. Вид этого пункта меню зависит от выделенного элемента модели.
- **Update** — позволяет вставить информацию обратного проектирования из внешнего файла с расширением red в разрабатываемую модель.
- **Print** — позволяет распечатать на принтере отдельные диаграммы и спецификации различных элементов разрабатываемой модели. В этом случае вызывается диалоговое окно выбора диаграмм и спецификаций для печати на подключенном к данному компьютеру принтеру.
- **Print Setup** — вызывается стандартное диалоговое окно макета страницы для настройки свойств печати.
- **Edit Path Map** — вызывает окно задания путей доступа к файлам системы IBM Rational Rose 2002. Как правило, значения путей, установленные по умолчанию, следует изменять только в случае крайней необходимости.
- Секция с именами последних файлов, с которыми осуществлялась работа в IBM Rational Rose 2002.
- **Exit** — прекращает работу и закрывает IBM Rational Rose 2002.

❒ Пункт меню **Edit** (Редактирование) содержит следующие операции (рис. 12.10).

- **Undo** — отменяет выполнение последнего действия по удалению или перемещению элементов модели.

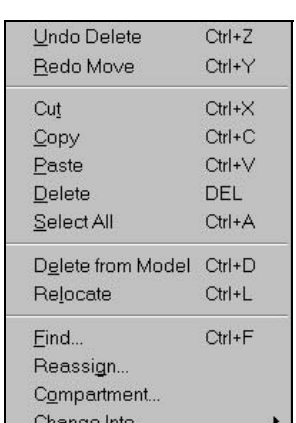


Рис. 12.10. Операции пункта меню **Edit** (Редактирование) главного меню

- **Redo** — повторяет действие после его отмены.
- **Cut** — вырезает выделенный элемент разрабатываемой модели и помещает его в буфер обмена.
- **Copy** — копирует выделенный элемент разрабатываемой модели и помещает его в буфер обмена.
- **Paste** — вставляет элемент разрабатываемой модели или его копию из буфера обмена в текущую активную диаграмму.
- **Delete** — удаляет выделенные элементы из текущей диаграммы, но не из разрабатываемой модели.
- **Select All** — выделяет все элементы на текущей диаграмме разрабатываемой модели.
- **Delete from Model** — удаляет все выделенные элементы из разрабатываемой модели.
- **Relocate** — позволяет перемещать или отменить перемещение классов, ассоциаций или компонентов из одного пакета в другой.
- **Find** — вызывает диалоговое меню поиска элемента в разрабатываемой модели по его имени.
- **Reassign** — позволяет заменить выделенный элемент разрабатываемой модели другим элементом модели.
- **Compartment** — позволяет отображать дополнительную информацию об объектах, классах, актерах или пакетах.
- **Change Info** — позволяет изменить тип выделенного элемента на текущей диаграмме на другой тип элемента.

❒ Пункт меню **View** (Вид) позволяет отображать на экране различные элементы рабочего интерфейса и изменять графическое представление диаграмм, содержит следующие операции (рис. 12.11).

- **Toolbars** — позволяет настроить внешний вид рабочего интерфейса системы и содержит дополнительные подпункты: **Standard** — делает видимой/невидимой стандартную панель инструментов (рис. 12.3); **Toolbox** — делает видимой/невидимой специальную панель инструментов текущей активной диаграммы (рис. 12.3); **Configure** — вызывает диалоговое окно настройки параметров модели, открытое на вкладке настройки панелей инструментов.
- **Status Bar** — делает видимой/невидимой строку состояния.
- **Documentation** — делает видимым/невидимым окно документации.
- **Browser** — делает видимым/невидимым браузер проекта.
- **Log** — делает видимым/невидимым окно журнала.
- **Editor** — делает видимым/невидимым встроенный текстовый редактор.

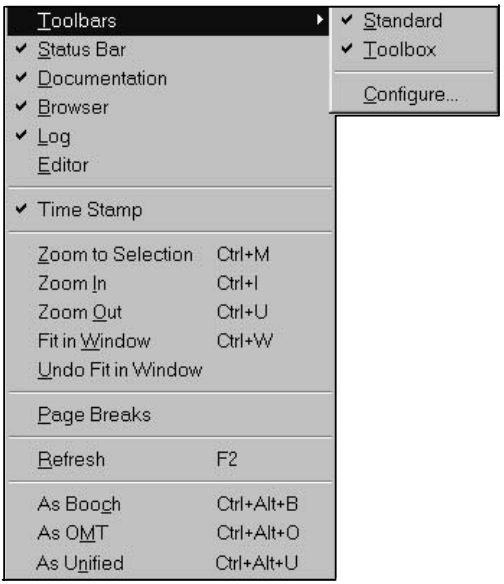


Рис. 12.11. Операции пункта меню **View** (Вид) главного меню

- **Time Stamp** — включает/выключает режим отображения времени в записях журнала.
- **Zoom to Selection** — изменяет масштаб изображения выделенных элементов модели, так чтобы они разместились в одном окне.
- **Zoom In** — увеличивает масштаб изображения.
- **Zoom Out** — уменьшает масштаб изображения.
- **Fit in Window** — изменяет (уменьшает) масштаб изображения всех элементов текущей диаграммы, так чтобы все они разместились в одном окне.
- **Undo Fit in Window** — отменяет изменение масштаба изображения для размещения элементов в одном окне.
- **Page Breaks** — разбивает текущую диаграмму на страницы для последующей печати.
- **Refresh** — перерисовывает текущую диаграмму.
- **As Booch** — изображает элементы модели в соответствии с нотацией Г. Буча.
- **As OMT** — изображает элементы модели в соответствии с нотацией OMT.
- **As Unified** — изображает элементы модели в соответствии с нотацией языка UML.

- Пункт меню **Format** (Формат) содержит следующие операции (рис. 12.12).
  - **Font Size** — изменяет размер используемого шрифта.
  - **Font** — вызывает диалоговое окно выбора шрифта.
  - **Line Color** — вызывает диалоговое окно выбора цвета линий.
  - **Fill Color** — вызывает диалоговое окно для выбора цвета заливки элементов диаграмм.
  - **Use Fill Color** — включает/выключает режим отображения цвета заливки элементов диаграмм.
  - **Automatic Resize** — включает/выключает режим автоматического изменения размеров графических элементов диаграмм для отображения текстовой информации об их свойствах.
  - **Stereotype Display** — позволяет выбрать способ изображения стереотипов выделенных элементов диаграммы и содержит дополнительные подпункты: **None** — стереотип не показывается; **Label** — стереотип отображается в форме текста; **Decoration** — стереотип отображается в форме небольшой пиктограммы в правом верхнем углу графического элемента; **Icon** — графический элемент диаграммы отображается в форме специального стереотипа.

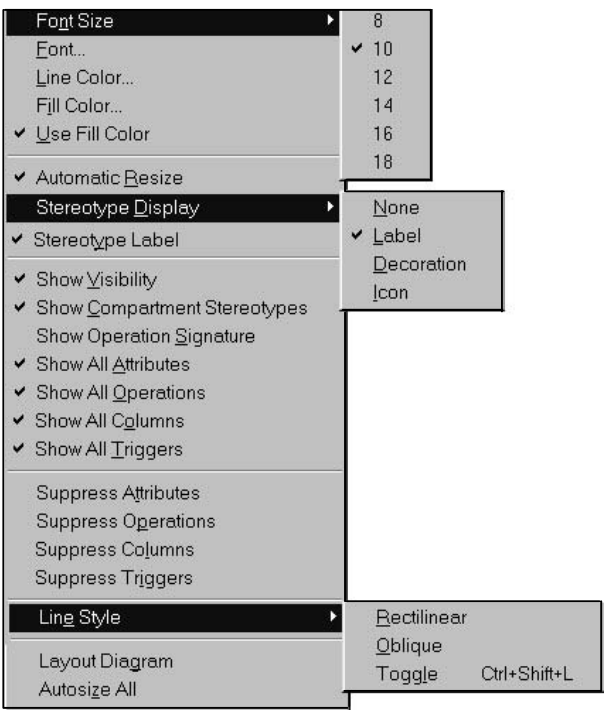


Рис. 12.12. Операции пункта меню **Format** (Формат) главного меню

- **Stereotype Label** — включает/выключает режим отображения текстовых стереотипов для взаимосвязей (ассоциаций, зависимостей и пр.) диаграммы.
- **Show Visibility** — включает/выключает режим отображения кванторов видимости атрибутов и операций классов.
- **Show Compartment Stereotypes** — включает/выключает режим отображения текстовых стереотипов атрибутов и операций классов.
- **Show Operation Signature** — включает/выключает режим отображения сигнатуры операций классов.
- **Show All Attributes** — включает/выключает режим отображения текстовых стереотипов атрибутов и операций классов.
- **Show All Operations** — делает видимыми/невидимыми операции классов.
- **Show All Columns** — делает видимыми/невидимыми столбцы (поля) таблицы модели данных.
- **Show All Triggers** — делает видимыми/невидимыми триггеры таблицы модели данных.
- **Suppress Attributes** — делает видимой/невидимой секцию атрибутов классов.
- **Suppress Operations** — делает видимой/невидимой секцию операций классов.
- **Suppress Columns** — делает видимой/невидимой секцию столбцов таблицы модели данных.
- **Suppress Triggers** — делает видимой/невидимой секцию триггеров таблицы модели данных.
- **Line Style** — позволяет выбрать способ графического изображения линий взаимосвязей и содержит дополнительные подпункты: **Rectilinear** — линия изображается в форме вертикальных и горизонтальных отрезков; **Oblique** — линия изображается в форме наклонных отрезков; **Toggle** — промежуточный вариант изображения линии.
- **Layout Diagram** — позволяет автоматически разместить графические элементы в окне диаграммы с минимальным количеством пересечений и наложений соединительных линий.
- **Autosize All** — позволяет автоматически изменить размеры графических элементов текущей диаграммы таким образом, чтобы текстовая информация помещалась внутри изображений соответствующих элементов.

❒ Пункт меню **Browse** (Обзор) содержит следующие операции (рис. 12.13).

- **Use Case Diagram** — вызывает диалоговое окно с предложением выбрать для отображения в рабочем окне одну из существующих диаграмм вариантов использования модели или приступить к разработке новой диаграммы.

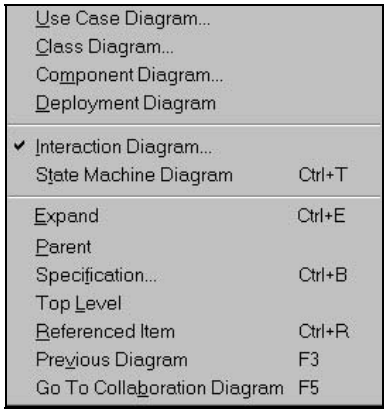


Рис. 12.13. Операции пункта меню **Browse** (Обзор) главного меню

- **Class Diagram** — вызывает диалоговое окно с предложением выбрать для отображения в рабочем окне одну из существующих диаграмм классов модели или приступить к разработке новой диаграммы.
- **Component Diagram** — вызывает диалоговое окно с предложением выбрать для отображения в рабочем окне одну из существующих диаграмм компонентов модели или приступить к разработке новой диаграммы.
- **Deployment Diagram** — вызывает диалоговое окно с предложением выбрать для отображения в рабочем окне одну из существующих диаграмм развертывания модели или приступить к разработке новой диаграммы.
- **Interaction Diagram** — вызывает диалоговое окно с предложением выбрать для отображения в рабочем окне одну из существующих диаграмм кооперации (последовательности) модели или приступить к разработке новой диаграммы.
- **State Machine Diagram** — вызывает диалоговое окно с предложением выбрать для отображения в рабочем окне одну из существующих диаграмм состояний модели или приступить к разработке новой диаграммы.
- **Expand** — отображает в рабочем окне первую из диаграмм выделенного пакета модели.
- **Parent** — отображает в рабочем окне родителя выделенной диаграммы модели.
- **Specification** — вызывает диалоговое окно свойств выделенного элемента модели.
- **Top Level** — отображает в рабочем окне диаграмму самого верхнего уровня для текущей диаграммы модели.

- **Referenced Item** — отображает в рабочем окне диаграмму классов, содержащую класс для выделенного объекта модели.
- **Previous Diagram** — отображает в рабочем окне предыдущую диаграмму модели.
- **Go To Collaboration Diagram (Go To Sequence Diagram)** — позволяет переключиться с представления диаграммы последовательности к представлению диаграммы кооперации и наоборот.

❒ Пункт меню **Report** (Отчет) содержит следующие операции (рис. 12.14).

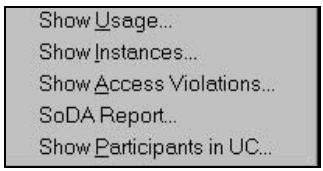


Рис. 12.14. Операции пункта меню **Report** (Отчет) главного меню

- **Show Usage** — отображает в диалоговом окне информацию об использовании выделенного элемента модели на различных диаграммах.
- **Show Instances** — отображает в диалоговом окне информацию об использовании объектов выделенного класса модели на различных диаграммах.
- **Show Access Violations** — отображает в диалоговом окне информацию о ссылках классов одного пакета на классы другого, при отсутствии соответствующей зависимости доступа (импорта) между этими пакетами модели.
- **SoDA Report** — позволяет сгенерировать отчет о разрабатываемой модели в формате MS Word с использованием специального средства IBM Rational SoDA.
- **Show Participants in UC** — отображает в диалоговом окне информацию о классах, компонентах и операциях, которые участвуют в реализации выделенного варианта использования модели на различных диаграммах.

❒ Пункт меню **Query** (Запрос) содержит следующие операции (рис. 12.15).

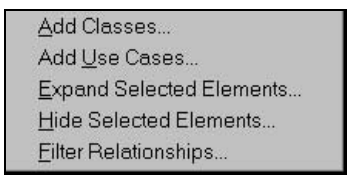


Рис. 12.15. Операции пункта меню **Query** (Запрос) главного меню

- **Add Classes** — вызывает диалоговое окно с предложением добавить на текущую диаграмму классы, которые имеются в модели на различных диаграммах.
- **Add Use Cases** — вызывает диалоговое окно с предложением добавить на текущую диаграмму варианты использования, которые имеются в модели на различных диаграммах.
- **Expand Selected Elements** — вызывает диалоговое окно с предложением добавить на текущую диаграмму элементы модели, которые связаны с выделенным элементом на других диаграммах.
- **Hide Selected Elements** — вызывает диалоговое окно с предложением удалить с текущей диаграммы элементы модели, которые связаны с выделенным элементом.
- **Filter Relationships** — вызывает диалоговое окно, позволяющее включить/выключить режим отображения различных отношений на текущей диаграмме.

❒ Пункт меню **Tools** (Инструменты) зависит от установленных на конкретное средство IBM Rational Rose 2002 расширений и может содержать, например, следующие операции (рис. 12.16).

- **Create** — создает новый элемент модели из предлагаемого списка для последующего размещения его на диаграмме, дублирует нажатие соответствующей кнопки на специальной панели инструментов.
- **Check Model** — проверяет разрабатываемую модель на наличие ошибок, информация о которых отображается в окне журнала.
- **Model Properties** — позволяет выполнить настройку свойств языка реализации выделенного элемента модели и содержит дополнительные подпункты: **Edit** — редактирование набора свойств; **View** — просмотр набора свойств; **Replace** — замена существующего набора свойств на новый набор, загружаемый из внешнего файла с расширением prp или pty; **Export** — сохранение существующего набора в файле с расширением prp или pty; **Add** — дополнение существующего набора свойств новым набором свойств, загружаемым из внешнего файла с расширением prp или pty; **Update** — обновляет существующий набор свойств после его редактирования или дополнения.
- **Options** — вызывает диалоговое окно настройки параметров модели, открытое на вкладке **General**.
- **Open Script** — вызывает стандартное диалоговое окно открытия внешнего файла, содержащего текст скрипта (файл с расширением ebs) для его редактирования в окне специального редактора скриптов.
- **New Script** — открывает дополнительное окно специального редактора скриптов для создания, отладки, выполнения и сохранения нового скрипта во внешнем файле с расширением ebs.

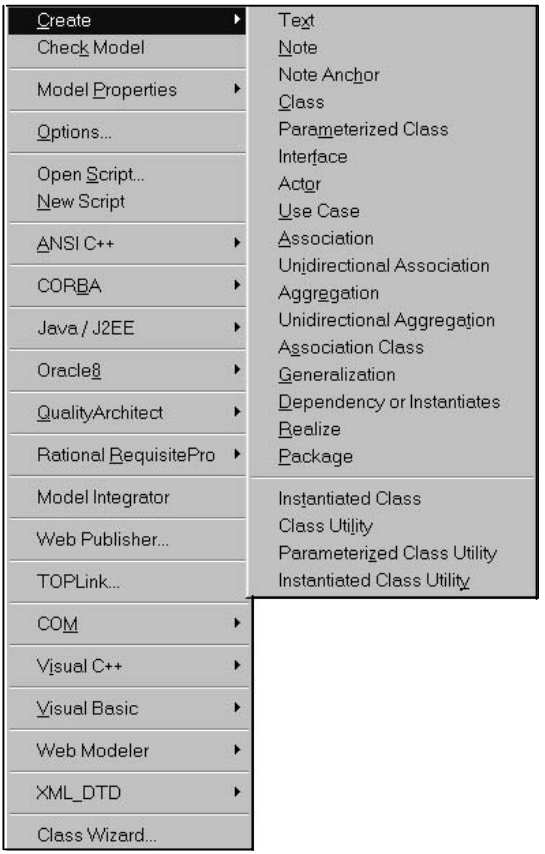


Рис. 12.16. Операции пункта меню **Tools** (Инструменты) главного меню

- **ANSI C++** — позволяет выполнить настройку свойств языка программирования C++ стандарта ANSI, выбранного в качестве языка реализации отдельных элементов модели.
- **CORBA** — позволяет выполнить настройку свойств и спецификацию модели для генерации объектов CORBA для реализации отдельных элементов модели.
- **Java/J2EE** — позволяет выполнить настройку свойств языка программирования Java, выбранного в качестве языка реализации отдельных элементов модели.
- **Oracle8** — позволяет выполнить настройку свойств и спецификацию модели для генерации схем СУБД Oracle 8 для отдельных элементов модели.
- **QualityArchitect** — позволяет выполнить настройку свойств и тестирование модели с помощью специального средства IBM Rational QualityArchitect.

- **Rational Requisite Pro** — позволяет выполнить настройку свойств модели для установления связей со специальным средством спецификации и управления требованиями.
- **Model Integrator** — открывает рабочее окно специального средства для интеграции модели.
- **Web Publisher** — позволяет выполнить настройку свойств модели для ее публикации в гипертекстовом формате.
- **TOPLink** — вызывает мастер преобразования таблиц модели данных в классы языка программирования Java, выбранного в качестве языка реализации отдельных элементов модели.
- **COM** — позволяет выполнить настройку свойств и спецификацию модели для генерации объектов COM с целью реализации отдельных элементов модели.
- **Visual C++** — позволяет выполнить настройку свойств и спецификацию модели для генерации программного кода MS Visual C++, выбранного в качестве языка реализации отдельных элементов модели.
- **Visual Basic** — позволяет выполнить настройку свойств и спецификацию модели для генерации программного кода MS Visual Basic, выбранного в качестве языка реализации отдельных элементов модели.
- **Web Modeler** — вызывает мастер преобразования существующего Web-узла в модель IBM Rational Rose 2002.
- **XML_DTD** — позволяет выполнить настройку свойств и спецификацию модели для ее публикации в формате расширяемого языка разметки XML.

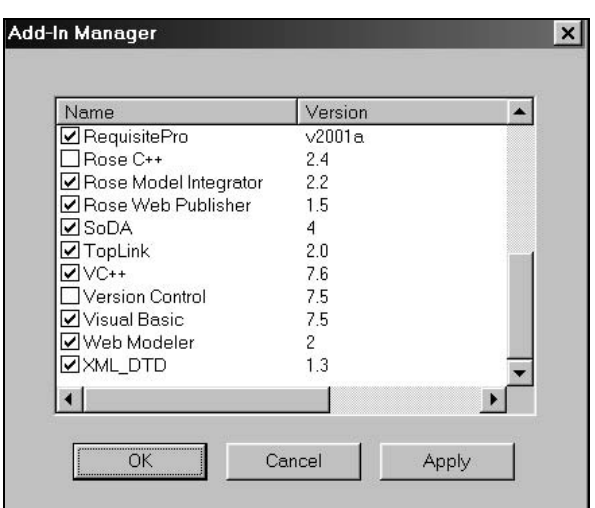


Рис. 12.17. Диалоговое окно **Add-In Manager** (Расширения) главного меню

- **Class Wizard** — вызывает мастер создания нового класса и его размещения на выбранной диаграмме модели.

❒ При выборе пункта главного меню **Add-Ins** (Расширения) вызывается диалоговое окно **Add-In Manager** для выбора расширений, из установленных при инсталляции средства IBM Rational Rose 2002 (рис. 12.17).

❒ Пункт меню **Window** (Окно) позволяет управлять окнами диаграмм и содержит следующие операции (рис. 12.18).

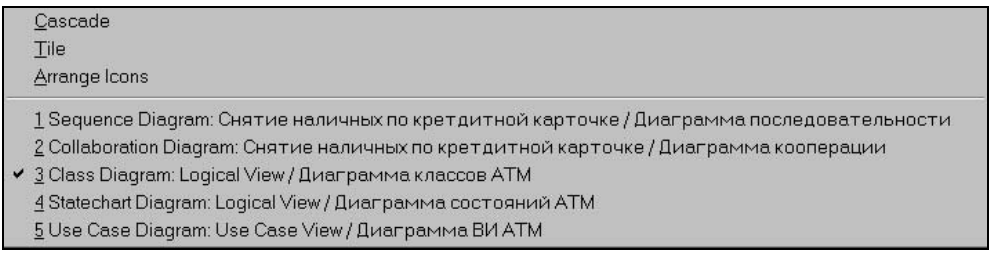


Рис. 12.18. Операции пункта меню **Window** (Окно) главного меню

- **Cascade** — каскадно размещает все открытые диаграммы модели.
- **Tile** — отображает все открытые диаграммы модели.
- **Arrange Icons** — упорядочивает расположение всех открытых диаграмм.
- Секция, содержащая имена всех открытых диаграмм модели для переключения между ними. Если открывается новая диаграмма, то в этой секции появляется новая строка с именем этой диаграммы и ее типом, выбрав которую, можно сразу перейти в нужное окно.

❒ Пункт меню **Help** (Справка) содержит следующие операции (рис. 12.19).

- **Contents and Index** — вызывает программу просмотра справочной системы, открытой на вкладке **Содержание**.
- **Search for Help On** — вызывает программу просмотра справочной системы, открытой на вкладке **Указатель**.

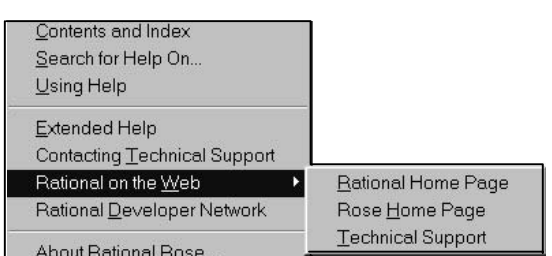


Рис. 12.19. Операции пункта меню **Help** (Справка) главного меню

- **Using Help** — вызывает программу просмотра справочной системы MS Windows.
- **Extended Help** — вызывает специальную программу расширенной справочной системы.
- **Contacting Technical Support** — вызывает установленный в операционной системе по умолчанию браузер и делает попытку соединиться с Web-сайтом технической поддержки компании IBM Rational (в случае наличия доступа в Интернет).
- **Rational on the Web** — вызывает установленный в операционной системе по умолчанию браузер и делает попытку соединиться с Web-сайтом компании IBM Rational (в случае наличия доступа в Интернет). Выбор отдельной операции этого пункта меню определяет загрузку той или иной Web-страницы компании, предназначенной для выполнения специальных действий по дополнительной поддержке средства IBM Rational Rose или загрузке имеющихся обновлений.
- **Rational Developer Network** — вызывает установленный в операционной системе по умолчанию браузер и делает попытку соединиться с Web-сайтом разработчиков компании IBM Rational (в случае наличия доступа в Интернет).
- **About Rational Rose** — отображает информацию о текущей рабочей версии IBM Rational Rose 2002.

12.4. Назначение операций стандартной панели инструментов

Стандартная панель инструментов содержит набор кнопок, которые дублируют наиболее часто выполняемые операции главного меню. Рассмотрим назначение отдельных кнопок панели инструментов (табл. 12.1).

Таблица 12.1. *Назначение кнопок стандартной панели инструментов*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Create New Model or File	Создает новую модель IBM Rational Rose 2002, которой по умолчанию присваивается имя untitled
	Open Existing Model or File	Вызывает стандартное диалоговое окно открытия внешнего файла с диска (файла с расширением mdl или ptl)
	Save Model, File or Script	Сохраняет разрабатываемую модель или скрипт во внешнем файле с соответствующим расширением

Таблица 12.1 *(продолжение)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Cut	Вырезает выделенные элементы модели и помещает их в буфер обмена
	Copy Diagram	Копирует выделенные элементы и помещает их в буфер обмена
	Paste	Вставляет элементы модели из буфера обмена в текущую диаграмму
	Print	Печатает текущую диаграмму на подключенном к компьютеру принтере
	Context Sensitive Help	Позволяет вызвать окно с контекстно-зависимой справкой для выбранного элемента модели
	View Documentation	Делает видимым/невидимым окно документации
	Browse Class Diagram	Позволяет быстро переключиться в окно диаграммы классов модели
	Browse Interaction Diagram	Позволяет быстро переключиться в окно диаграммы кооперации (последовательности) модели
	Browse Component Diagram	Позволяет быстро переключиться в окно диаграммы компонентов модели
	Browse State Machine Diagram	Позволяет быстро переключиться в окно диаграммы состояний модели
	Browse Deployment Diagram	Позволяет быстро переключиться в окно диаграммы развертывания модели
	Browse Parent	Позволяет быстро переключиться в окно диаграммы модели с родителем текущей диаграммы
	Browse Previous Diagram	Позволяет быстро переключиться в окно предыдущей диаграммы модели
	Zoom In	Увеличивает масштаб изображения элементов текущей диаграммы модели
	Zoom Out	Уменьшает масштаб изображения элементов текущей диаграммы модели

Таблица 12.1 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Fit in Window	Уменьшает масштаб изображения элементов текущей диаграммы модели для ее размещения полностью в окне диаграммы
	Undo Fit in Window	Отменяет уменьшение масштаба изображения элементов текущей диаграммы модели после ее размещения полностью в окне диаграммы

Одним из наиболее интересных характеристик средства IBM Rational Rose 2002 является возможность генерации программного кода на основе построенной модели. Как уже отмечалось ранее, возможность генерации текста программы на том или ином языке программирования зависит от установленных расширений IBM Rational Rose 2002. Пример генерации текста программного кода на языке C++ рассматривается далее в *главе 14*.

Глава 13

Начало работы над проектом в среде IBM Rational Rose 2002

Процесс работы над проектом состоит в последовательной разработке канонических диаграмм, которые в совокупности представляют интегрированную модель разрабатываемой программной системы. Хотя последовательность разработки диаграмм в языке UML не определена, для этой цели можно воспользоваться рекомендациями рационального унифицированного процесса RUP. Принятый в книге порядок разработки апробирован на нескольких реальных проектах и оказался довольно удачным.

Концептуально процесс разработки канонических диаграмм заключается в размещении на диаграммах соответствующих графических элементов, установлении отношений между этими элементами, их спецификации и документировании, в соответствии с рассмотренными в *части II* правилами и нотацией языка UML. После построения модели, проверки правильности и согласованности свойств ее элементов можно сгенерировать текст программного кода на одном из выбранных языков программирования. В последующем этот код следует доработать в той или иной среде программирования с целью получения исполнимых модулей программы, ориентированной на работу в определенной операционной среде и вычислительной платформе.

Технически процесс разработки графических диаграмм во многом аналогичен процессу работы в популярных средах визуального программирования. Разработчик выбирает необходимый графический элемент посредством нажатия соответствующей кнопки на специальной панели инструментов и размещает этот элемент на рабочем листе канонической диаграммы. После этого редактируется набор свойств этого элемента в соответствии с рассмотренной нотацией языка UML.

13.1. Разработка диаграммы вариантов использования в среде IBM Rational Rose

Работа над проектом начинается с общего анализа проблемы и построения диаграммы вариантов использования, которая отражает функциональное назначение проектируемой программной системы. Общие рекомендации

по построению диаграммы вариантов использования были рассмотрены в *главе 4*. В качестве разрабатываемого проекта рассматривается модель системы управления банкоматом, которая использовалась в качестве сквозного примера при иллюстрации элементов нотации языка UML в *части II*.

Для вновь создаваемого проекта можно воспользоваться мастером типовых проектов, если он установлен в данной конфигурации. Мастер типовых проектов доступен из меню **File → New** (Файл → Создать) или при загрузке средства IBM Rational Rose 2002.

Примечание

В нашем случае следует отказаться от мастера, в результате чего появится рабочий интерфейс с чистым окном активной диаграммы классов и именем проекта `untitled` по умолчанию. Для изменения имени проекта следует сохранить модель в файле на диске, например, под именем `ATM_Model.mdl`. В этом случае изменится имя в строке заголовка и имя папки проекта в иерархическом представлении модели в браузере проекта.

Как и другие программы, средство IBM Rational Rose позволяет настраивать глобальные параметры среды, такие как выбор шрифтов и цвета для представления различных элементов модели. Настройка шрифтов, цвета линий и графических элементов производится через операцию главного меню: **Tools → Options** (Инструменты → Параметры).

Примечание

Характерной особенностью среды является возможность работы с символами кириллицы. Однако следует заметить, что при спецификации элементов модели с последующей генерацией текста программного кода нужно сразу записывать имена и свойства классов, ассоциаций, атрибутов, операций и компонентов символами того языка, который поддерживается соответствующим языком программирования.

Для разработки диаграммы вариантов использования модели необходимо активизировать соответствующую диаграмму в окне диаграммы. Это можно сделать различными способами.

- Раскрыть представление вариантов использования в браузере (Use Case View) и дважды щелкнуть на пиктограмме **Main** (Главная).
- Через пункт меню **Browse → Use Case Diagram** (Обзор → Диаграмма вариантов использования).
- Нажав соответствующую кнопку на стандартной панели инструментов (см. табл. 12.1).

При этом появляется новое окно с чистым рабочим листом диаграммы вариантов использования и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необходимых для разра-

ботки диаграммы вариантов использования (табл. 13.1). Назначение отдельных кнопок панели можно узнать из всплывающих подсказок.

Таблица 13.1. *Назначение кнопок специальной панели инструментов диаграммы вариантов использования*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	Package	Добавляет на диаграмму пакет
	Use Case	Добавляет на диаграмму вариант использования
	Actor	Добавляет на диаграмму актера
	Unidirectional Association	Добавляет на диаграмму направленную ассоциацию
	Dependency or instantiates	Добавляет на диаграмму отношение зависимости
	Generalization	Добавляет на диаграмму отношение обобщения

На специальной панели инструментов по умолчанию присутствует только часть пиктограмм элементов, которые могут быть использованы для построения диаграммы. Добавить кнопки с пиктограммами других графических элементов, например, таких как бизнес-вариант использования (business use case), бизнес-актер (business actor), сотрудник (business worker), или удалить ненужные кнопки можно с помощью настройки специальной панели инструментов. Диалоговое окно настройки удобно вызвать с помощью пункта контекстного меню **Customize** (Настройка) при позиционировании курсора на специальной панели инструментов (рис. 13.1).

Для добавления необходимых кнопок на панель следует выделить их в левом окне со списком пиктограмм графических элементов, после чего нажать

кнопку **Добавить** в центре диалогового окна. Для удаления ненужных кнопок с панели инструментов следует выделить их в правом окне со списком пиктограмм графических элементов, после чего нажать кнопку **Удалить** в центре диалогового окна. Для восстановления набора пиктограмм по умолчанию можно нажать кнопку **Сброс**. После настройки специальной панели инструментов соответствующее окно следует закрыть нажатием на кнопку **Закрыть**.

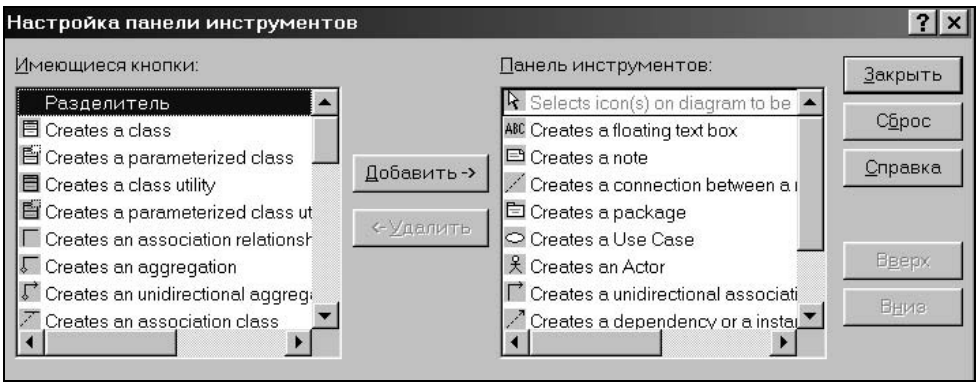


Рис. 13.1. Диалоговое окно настройки специальной панели инструментов диаграммы вариантов использования

Примечание

Открыть диалоговое окно настройки специальных панелей инструментов диаграмм можно с помощью операции главного меню **Tools** → **Options** (Инструменты → Параметры), раскрыв вкладку **Toolbars** (Панели инструментов) и нажав соответствующую кнопку (например, **Use Case diagram**) в группе опций **Customize Toolbars** (Настройка панелей инструментов).

Напомним, что в качестве разрабатываемого проекта служит модель системы управления банкоматом.

13.1.1. Добавление актера на диаграмму вариантов использования

Для добавления актера на диаграмму варианта использования нужно с помощью левой кнопки мыши нажать кнопку с изображением пиктограммы актера на специальной панели инструментов и щелкнуть левой кнопкой мыши на свободном месте рабочего листа диаграммы. На диаграмме появится изображение варианта использования с маркерами изменения его геометрических размеров и предложенным по умолчанию именем (рис. 13.2).

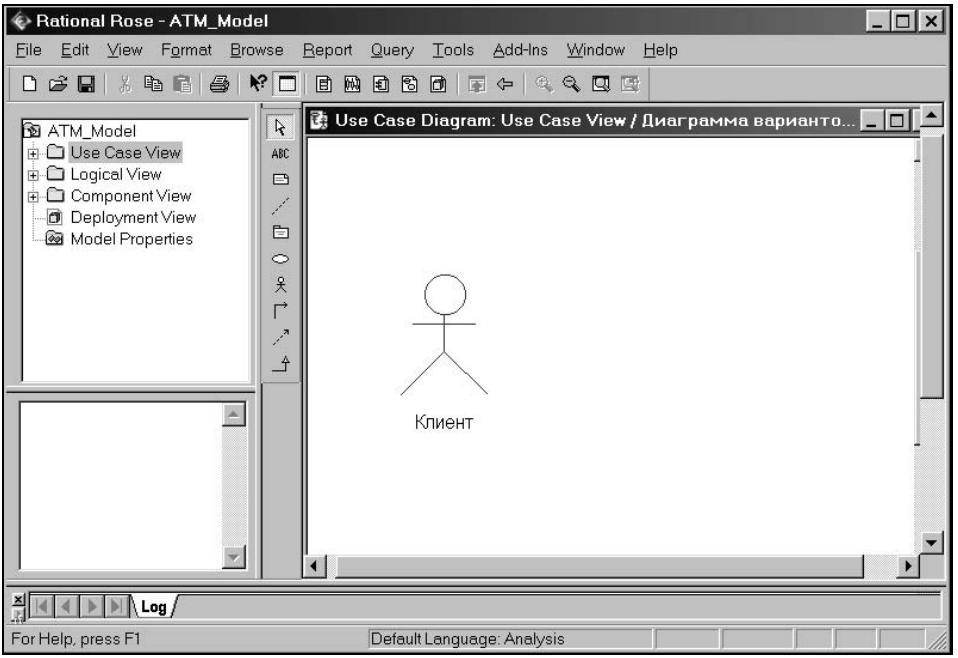


Рис. 13.2. Диаграмма вариантов использования после добавления на нее актера

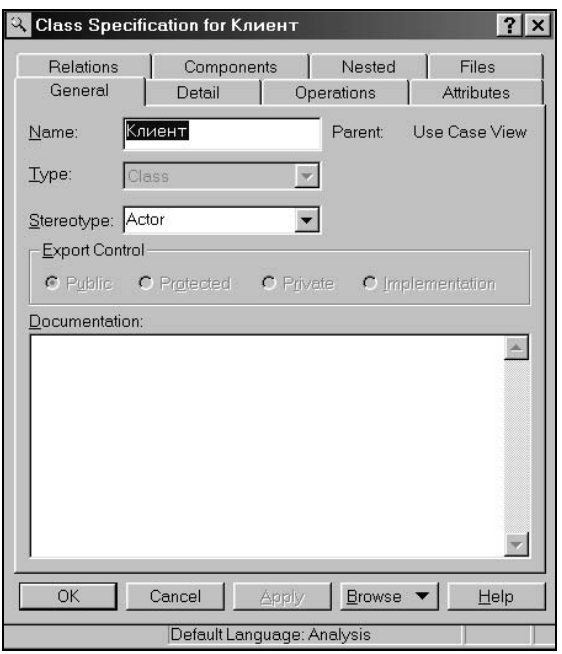


Рис. 13.3. Диалоговое окно настройки свойств актера

Имя добавленного элемента может быть изменено разработчиком либо сразу после размещения элемента на диаграмме, либо в ходе последующей работы над проектом. По щелчку правой кнопкой мыши на выбранном элементе вызывается контекстное меню элемента, среди опций которого имеется пункт **Open Specification** (Открыть спецификацию). В этом случае активизируется диалоговое окно со специальными вкладками, в поля которых можно занести всю информацию по данному актеру (рис. 13.3).

Примечание

Открыть диалоговое окно спецификации графического элемента можно также двойным щелчком левой кнопкой мыши на изображении этого элемента на диаграмме. Следует заметить, что в среде IBM Rational Rose актер является классом, что следует учитывать при спецификации его свойств. В частности, для актера некорректно специфицировать атрибуты и операции, поскольку он является внешней по отношению к разрабатываемой системе сущностью.

13.1.2. Добавление варианта использования

Для добавления варианта использования на диаграмму нужно, с помощью левой кнопки мыши, нажать кнопку с изображением варианта использования на специальной панели инструментов и щелкнуть левой кнопкой мыши на свободном месте диаграммы. На диаграмме появится изображение варианта использования с маркерами изменения его геометрических размеров и предложенным по умолчанию именем (рис. 13.4).

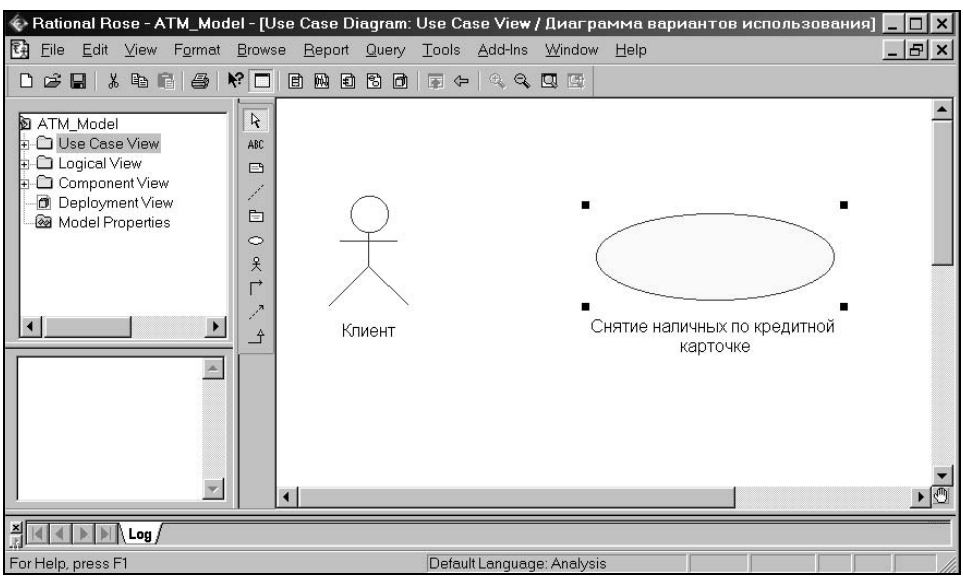


Рис. 13.4. Диаграмма вариантов использования после добавления на нее варианта использования

Для варианта использования можно также открыть диалоговое окно спецификации двойным щелчком левой кнопкой мыши на изображении этого элемента.

13.1.3. Добавление ассоциации

Для добавления ассоциации между актером и вариантом использования на диаграмму нужно, с помощью левой кнопки мыши, нажать на специальной панели инструментов кнопку с изображением пиктограммы направленной ассоциации, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении актера на диаграмме и отпустить ее на изображении варианта использования. В результате этих действий на диаграмме появится изображение ассоциации, соединяющей актера с вариантом использования (рис. 13.5).

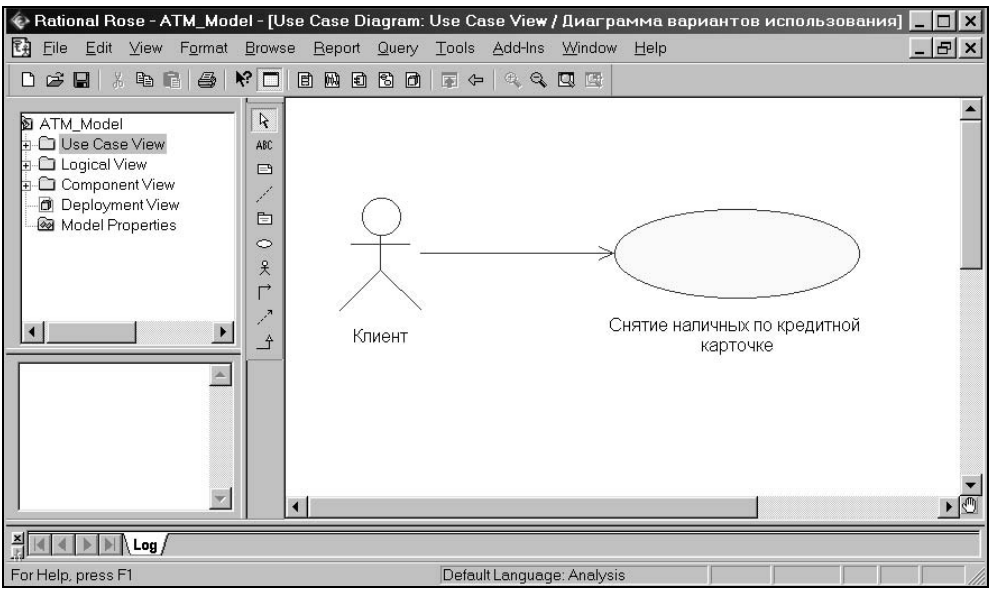


Рис. 13.5. Диаграмма вариантов использования после добавления на нее ассоциации

При необходимости можно сделать направленную ассоциацию ненаправленной, для чего следует воспользоваться диалоговым окном свойств ассоциации (рис. 13.6). Открыть это окно можно двойным щелчком на изображении линии ассоциации на диаграмме, после чего убрать отметку строки выбора **Navigable** (Навигация) на вкладке **Role A Detail** (Детальные свойства концевой точки ассоциации А).

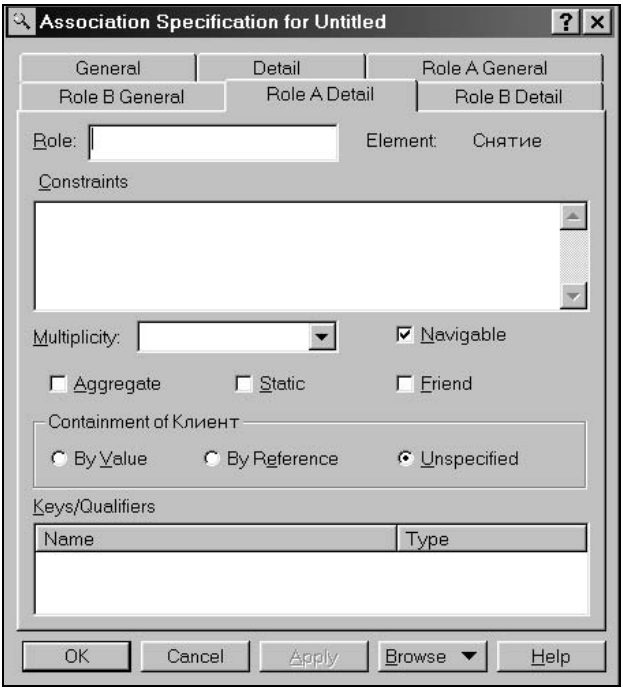


Рис. 13.6. Диалоговое окно настройки свойств ассоциации

Более подробно спецификация других свойств ассоциаций будет рассмотрена далее при разработке диаграммы классов в *подразделе 13.2.*

13.1.4. Добавление отношения зависимости

Для добавления отношения зависимости между двумя вариантами использования на диаграмму необходимо, предварительно, добавить вариант использования `Проверка ПИН-кода`. После чего нужно с помощью левой кнопки мыши нажать кнопку с изображением пиктограммы зависимости на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении варианта использования `Снятие наличных по кредитной карточке` на диаграмме и отпустить ее на изображении варианта использования `Проверка ПИН-кода`.

В результате этих действий на диаграмме появится изображение отношения зависимости, соединяющего два выбранных варианта использования (рис. 13.7).

Для указания дополнительного стереотипа <<include>> для добавленного отношения зависимости следует уже известным способом открыть диалоговое окно свойств этого отношения (рис. 13.8) и выбрать нужный стереотип из предлагаемого списка, после чего нажать кнопку **Apply** (Применить).

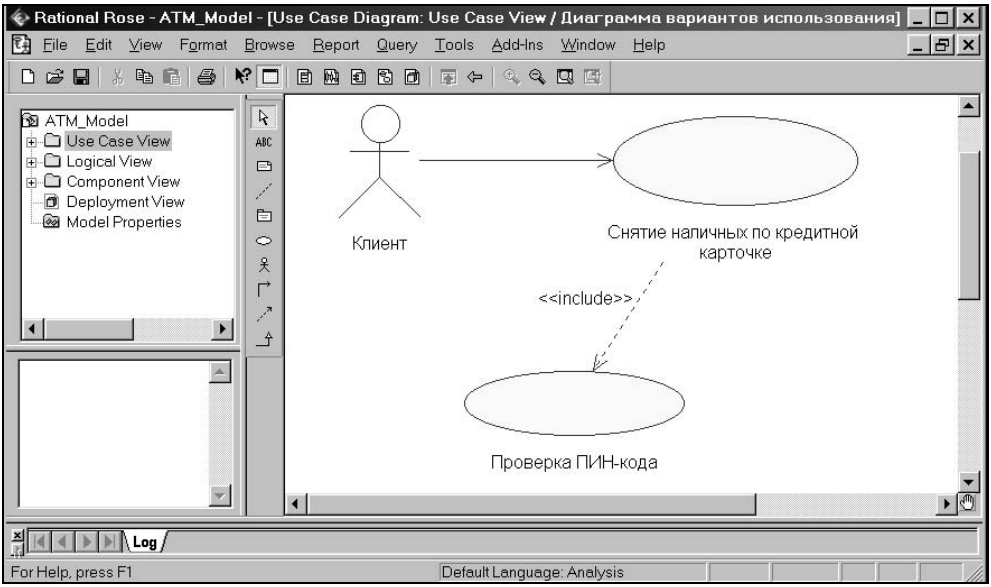


Рис. 13.7. Диаграмма вариантов использования после добавления на нее отношения зависимости

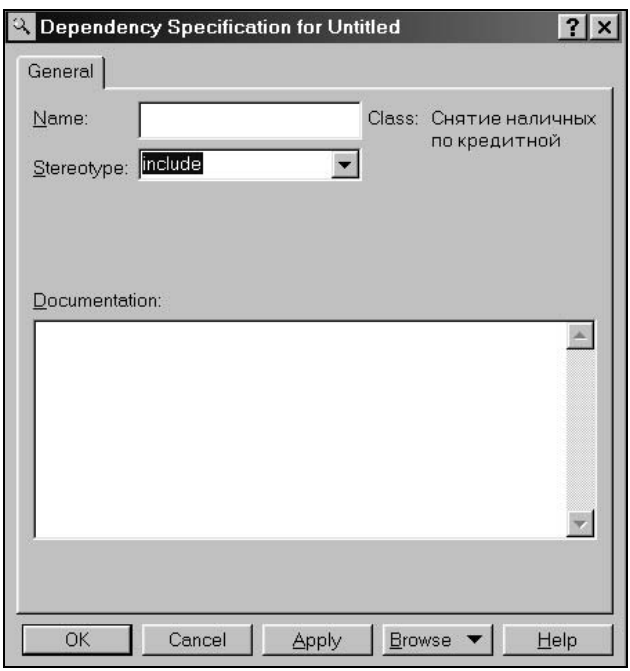


Рис. 13.8. Диалоговое окно настройки свойств отношения зависимости

При необходимости аналогичным образом могут быть добавлены на диаграмму вариантов использования отношения зависимости со стереотипом <<extend>>, которые применяются для моделирования исключений варианта использования.

13.1.5. Добавление текстового файла со сценарием варианта использования

Для добавления к варианту использования текстового файла (например, к варианту использования `Снятие наличных по кредитной карточке` — файла формата MS Word) с описанием сценария его выполнения (см. табл. 4.2–4.4) необходимо выделить этот вариант использования в браузере проекта, по щелчку правой кнопки мыши вызвать контекстное меню и выполнить операцию **New → File** (Новый → Файл). В результате этого будет вызвано стандартное окно открытия файла, в котором необходимо задать имя предварительно созданного в среде MS Word добавляемого файла.

После нажатия кнопки **Открыть** пиктограмма добавленного файла появится в браузере проекта ниже соответствующего варианта использования. В последующем, можно вернуться к редактированию этого файла сценария, выполнив двойной щелчок на этой пиктограмме. При этом файл сценария будет открыт в соответствующем приложении (в рассматриваемом случае — в текстовом процессоре MS Word).

Для окончательного построения диаграммы варианта использования рассматриваемого примера следует аналогичным образом добавить актера с именем `Банк`, вариант использования `Получение справки о состоянии счета`, а также соответствующие ассоциации и отношение зависимости. Построенная таким образом диаграмма вариантов использования будет иметь следующий вид (рис. 13.9).

Напомним, что диаграмма вариантов использования является высокоуровневым представлением модели, поэтому она не должна содержать слишком много вариантов использования и актеров. В последующем построенная диаграмма может быть изменена добавлением новых элементов, таких как варианты использования и актеров, или их удалением.

Для удаления графического элемента с диаграммы его следует выделить на диаграмме и нажать клавишу **Delete** на клавиатуре. При этом выделенный элемент будет удален с активной диаграммы, но не из модели. Для удаления элемента не только из диаграммы, но и из модели проекта необходимо выделить удаляемый элемент на диаграмме и воспользоваться пунктом меню **Edit → Delete from Model** (Редактирование → Удалить из модели). Для этой же цели служит комбинация клавиш быстрого доступа **Ctrl + D**.

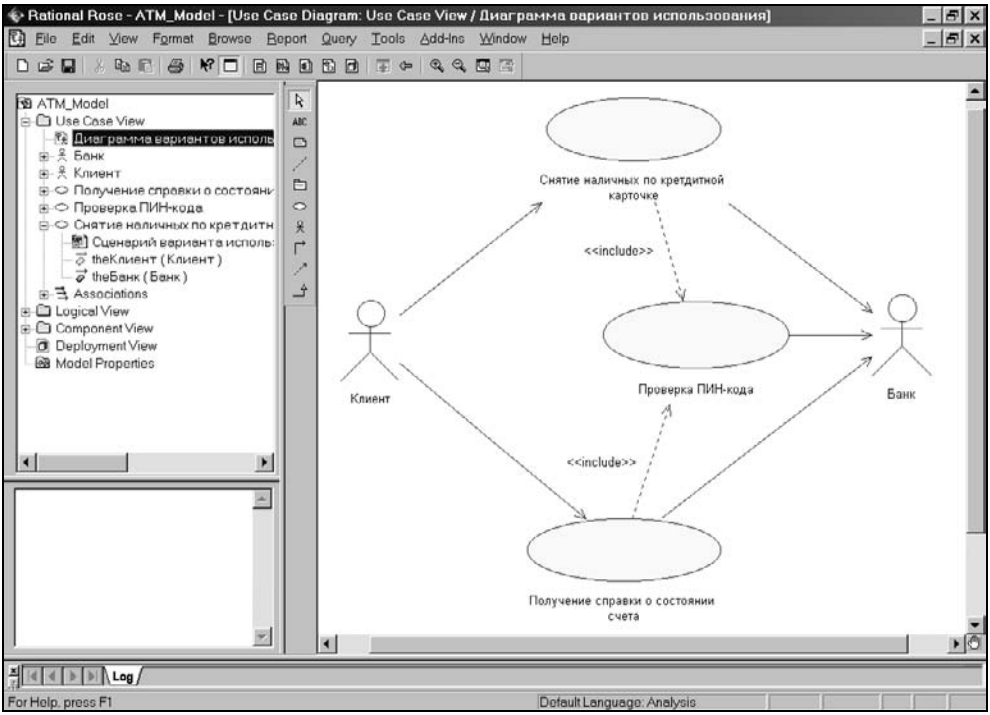


Рис. 13.9. Окончательный вид диаграммы вариантов использования разрабатываемой модели управления банкоматом

Примечание

При работе со связями на диаграмме вариантов использования следует помнить о назначении соответствующих связей в нотации языка UML. Речь идет о том, что если для двух элементов выбранный вид связи не является допустимым, то в большинстве случаев среда IBM Rational Rose 2002 сообщит об этом разработчику, и такая связь не будет добавлена на диаграмму. Разработку модели системы управления банкоматом продолжим построением следующего типа канонических диаграмм — диаграммы классов.

13.2. Разработка диаграммы классов в среде IBM Rational Rose

Диаграмма классов является основным логическим представлением модели и содержит детальную информацию о внутреннем устройстве объектно-ориентированной программной системы. Общие рекомендации по построению

диаграммы классов были рассмотрены в *главе 5*. Активизировать рабочее окно диаграммы классов можно несколькими способами.

- Окно диаграммы классов появляется по умолчанию в рабочем окне диаграммы после создания нового проекта.
- Щелкнуть на кнопке с изображением диаграммы классов на стандартной панели инструментов.
- Раскрыть логическое представление (Logical View) в браузере и дважды щелкнуть на пиктограмме **Main** (Главная).
- Выполнить операцию главного меню **Browse** → **Class Diagram** (Обзор → Диаграмма классов).

При этом появляется новое окно с чистым рабочим листом диаграммы классов и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необходимых для разработки диаграммы классов (табл. 13.2). Назначение отдельных кнопок панели можно узнать также из всплывающих подсказок.

Таблица 13.2. *Назначение кнопок специальной панели инструментов диаграммы классов*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	Package	Добавляет на диаграмму пакет
	Class	Добавляет на диаграмму вариант использования
	Interface	Добавляет на диаграмму актера
	Unidirectional Association	Добавляет на диаграмму направленную ассоциацию
	Association Class	Добавляет на диаграмму ассоциацию-класс

Таблица 13.2 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Dependency or instantiates	Добавляет на диаграмму отношение зависимости
	Generalization	Добавляет на диаграмму отношение обобщения
	Realize	Добавляет на диаграмму отношение реализации

На специальной панели инструментов по умолчанию присутствует только часть пиктограмм элементов, которые могут быть использованы для построения диаграммы классов. Добавить кнопки с пиктограммами других графических элементов, таких как, например, отношение агрегации, шаблон, класс бизнес-сущность, организационное подразделение, или удалить ненужные кнопки можно с помощью настройки специальной панели инструментов. При этом диалоговое окно настройки можно вызвать аналогично другим панелям, с помощью пункта контекстного меню **Customize** (Настройка) при позиционировании курсора на специальной панели инструментов.

13.2.1. Добавление класса на диаграмму классов

Для добавления класса на диаграмму классов нужно с помощью левой кнопки мыши нажать кнопку с изображением пиктограммы класса на специальной панели инструментов, отпустить левую кнопку мыши и щелкнуть левой кнопкой мыши на свободном месте рабочего листа диаграммы. На диаграмме появится изображение класса с маркерами изменения его геометрических размеров и предложенным по умолчанию именем, которое разработчику следует изменить (рис. 13.10).

Хотя для рассматриваемой модели системы управления банкоматом диаграмма классов была построена в *главе 5*, разрабатываемая диаграмма классов несколько отличается от ранее построенной (рис. 5.22), главным образом, формой записи имен ее графических элементов.

Примечание

Поскольку предполагается программная реализация разрабатываемой модели, имена классов, атрибутов и операций записаны символами латиницы. Чтобы не затруднять визуальное восприятие диаграмм, в этом случае можно воспользоваться способом простой транслитерации соответствующих имен на русском. Именно этот вариант выбран для представления диаграммы классов системы управления банкоматом в книге.

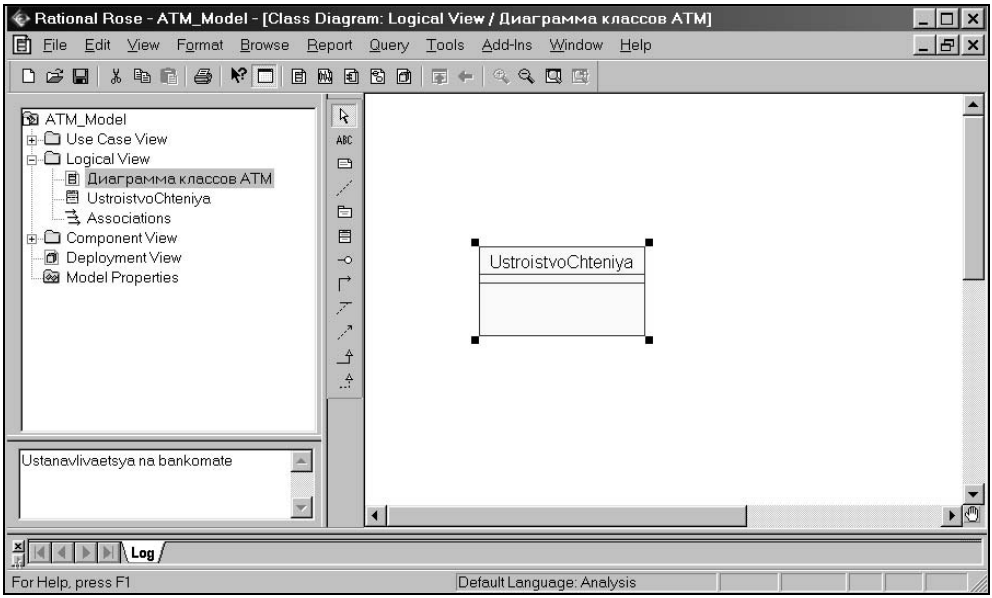


Рис. 13.10. Диаграмма классов после добавления на нее класса `УстройствоЧтения`

13.2.2. Добавление атрибутов классов

Из всех графических элементов класс обладает максимальным набором свойств, главными из которых являются его атрибуты и операции. Добавить атрибут к созданному ранее классу можно одним из следующих способов:

- с помощью операции контекстного меню **New Attribute** (Новый атрибут) для класса, выделенного на диаграмме классов. В этом случае активизируется курсор ввода в области графического изображения класса на диаграмме;
- с помощью операции контекстного меню **New → Attribute** (Новый → Атрибут) для класса, выделенного в браузере проекта. В этом случае активизируется курсор ввода в области иерархического представления класса в браузере под именем соответствующего класса;
- с помощью операции контекстного меню **Insert** (Вставить), вызванного при позиционировании курсора в области открытой вкладки атрибутов в диалоговом окне свойств **Class Specification** соответствующего класса (рис. 13.11). Напомним, что открыть диалоговое окно свойств класса можно двойным щелчком левой кнопкой мыши на изображении этого класса на диаграмме или в браузере проекта.

После добавления атрибута к классу ему присваивается по умолчанию имя `name` и некоторый квантор видимости. При этом видимость атрибутов

и операций изображается в форме специальных пиктограмм или украшений. Используемые пиктограммы видимости изображаются перед именем соответствующего атрибута или операции и имеют следующий смысл (табл. 13.3).

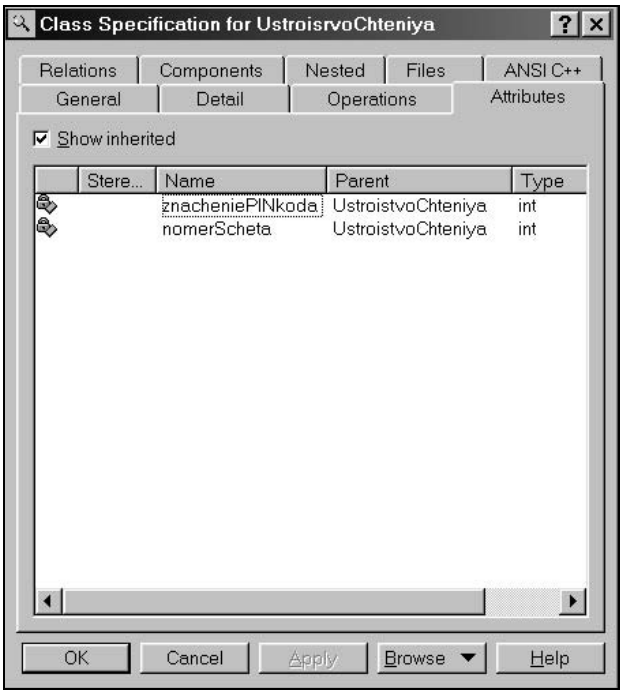


Рис. 13.11. Диалоговое окно настройки свойств класса, открытое на вкладке атрибутов

Таблица 13.3. *Пиктограммы видимости атрибутов и операций классов*

Графическое изображение	Текстовый аналог	Назначение пиктограммы
	Public	Общий, открытый. В нотации языка UML такому атрибуту соответствует знак "+"
	Protected	Защищенный. В нотации языка UML такому атрибуту соответствует знак "#"
	Private	Закрытый. В нотации языка UML такому атрибуту соответствует знак "-"
	Implementation	Пакетный. В нотации языка UML такому атрибуту соответствует знак "~"

Для атрибутов классов можно задавать также тип данных и начальные значения, а также назначить стереотип из раскрывающегося списка.

13.2.3. Добавление операций классов

Добавить операцию к созданному ранее классу можно одним из следующих способов:

- с помощью операции контекстного меню **New Operation** (Новая операция) для класса, выделенного на диаграмме классов. В этом случае активизируется курсор ввода в области графического изображения класса на диаграмме;
- с помощью операции контекстного меню **New → Operation** (Новая → Операция) для класса, выделенного в браузере проекта. В этом случае активизируется курсор ввода в области иерархического представления класса в браузере под именем соответствующего класса;
- с помощью операции контекстного меню **Insert** (Вставить), вызванного при позиционировании курсора в области открытой вкладки операций в диалоговом окне свойств **Class Specification** соответствующего класса.

После добавления операции к классу ей присваивается по умолчанию имя `opname` и некоторый квантор видимости. Используемые пиктограммы видимости операций изображаются аналогично видимости атрибутов (табл. 13.3). Каждая из операций классов имеет собственное диалоговое окно свойств **Operation Specification**, которое может быть открыто по двойному щелчку на имени операции, на соответствующей вкладке свойств класса или на имени этой операции в браузере проекта (рис. 13.12).

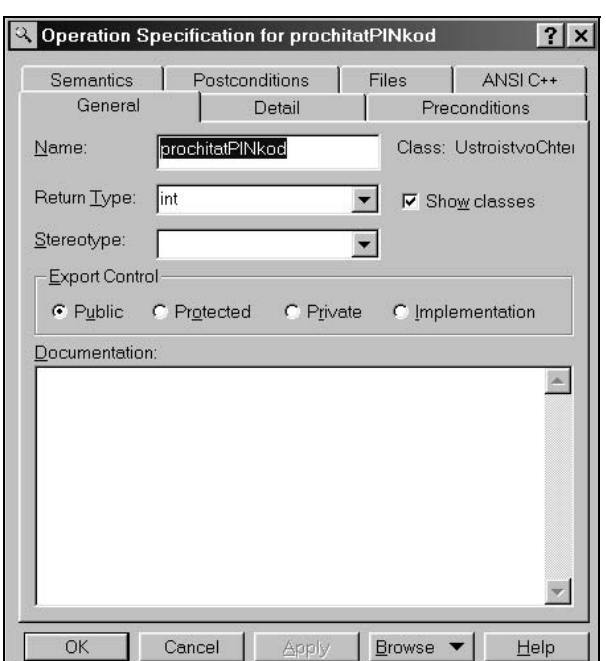


Рис. 13.12. Диалоговое окно настройки свойств операции класса

Для отдельных операций выбранного класса можно задавать: тип возвращаемого результата, аргументы и их тип, стереотип, а также определить протокол и размер, задать исключительные ситуации, специфицировать пред- и постусловий, а также целый ряд других свойств, детальное рассмотрение которых возможно только в контексте конкретно выбранного языка реализации проекта.

13.2.4. Добавление отношений на диаграмму классов

Добавление на диаграмму классов отношений между классами типа ассоциации, зависимости, агрегации, композиции, реализации и обобщения выполняется следующим образом. На специальной панели инструментов выбирается требуемый тип отношения щелчком по кнопке с соответствующим изображением. Если отношение направленное, то на диаграмме классов надо выделить первый элемент отношения (источник, от которого исходит стрелка) и, не отпуская нажатую левую кнопку мыши, переместить ее указатель ко второму элементу отношения (приемник, к которому направлена стрелка). После перемещения ко второму элементу кнопку мыши следует отпустить, а на диаграмму классов будет добавлено новое отношение.

Примечание

Следует заметить, что по умолчанию на специальной панели инструментов диаграммы классов может отсутствовать кнопка с пиктограммой агрегации. В этом случае необходимо предварительно добавить ее на панель инструментов описанным ранее способом (рис. 13.1). Впрочем, для изображения отношения агрегации можно вначале изобразить обычную ассоциацию, после чего, открыв окно ее свойств на вкладке деталей соответствующего конца ассоциации, выставить отметку в строке выбора **Aggregate** (Агрегация).

Для изображения отношения композиции можно сначала изобразить обычную ассоциацию, после чего, открыв окно ее свойств на вкладке деталей соответствующего конца ассоциации, выставить отметку в строке выбора **Aggregate** (Агрегация) и выбрать опцию **By Value** (По значению) в секции **Containment** (Включение атрибутов).

Если же отношение ненаправленное (двунаправленное), то порядок выбора классов для этого отношения произвольный. Для отношений можно определить кратность каждого из концов, задать имя и стереотип, использовать ограничения и роли, а также некоторые другие свойства. Доступ к диалоговому окну свойств отношений можно получить после выделения линии этого отношения на диаграмме классов или в браузере проекта и двойного щелчка левой кнопки мыши.

Для окончательного построения диаграммы классов рассматриваемого примера следует описанным ранее способом добавить оставшиеся классы и ассоциации, а также специфицировать стереотипы, атрибуты и операции этих классов. Построенная таким образом диаграмма классов будет иметь следующий вид (рис. 13.13).

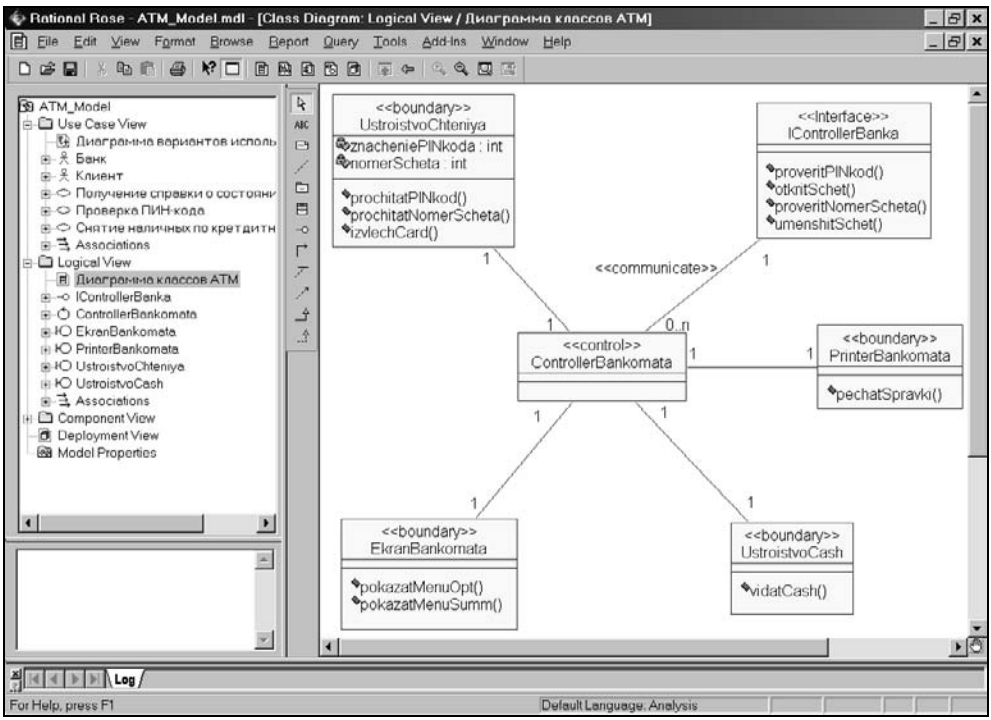


Рис. 13.13. Окончательный вид диаграммы классов разрабатываемой модели управления банкоматом

На изображенной диаграмме классов выбран текстовый способ изображения стереотипов классов, при котором стереотип записывается в угловых кавычках выше имени соответствующего класса. Средство IBM Rational Rose 2002 позволяет также изображать стереотипы в форме графических изображений (как в браузере проекта), в форме пиктограммы в верхней секции изображения класса на диаграмме или вообще отказаться от изображения стереотипов.

Изменить изображение стереотипа для выбранного класса можно, например, с помощью операции контекстного **Options → Stereotype Display** (Параметры → Изображение стереотипов). Выполнение этой операции можно предложить читателям в качестве упражнения.

13.3. Разработка диаграммы кооперации в среде IBM Rational Rose

Диаграмма кооперации является разновидностью диаграммы взаимодействия и в контексте языка UML описывает динамический аспект взаимодействия объектов при реализации отдельных вариантов использования. Общие рекомендации по построению диаграммы кооперации были рассмотрены в *главе 6*. Активизировать рабочее окно диаграммы кооперации можно несколькими способами:

- щелкнуть на кнопке с изображением диаграммы взаимодействия на стандартной панели инструментов и выбрать для построения диаграмму кооперации;
- выполнить операцию главного меню **Browse** → **Interaction Diagram** (Обзор → Диаграмма взаимодействия) и выбрать для построения диаграмму кооперации;
- выполнить операцию контекстного меню **New** → **Collaboration Diagram** (новая → Диаграмма кооперации) для логического представления или представления вариантов использования в браузере проекта.

При этом появляется новое окно с чистым рабочим листом диаграммы кооперации и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необходимых для разработки диаграммы кооперации (табл. 13.4). Назначение отдельных кнопок панели можно узнать из всплывающих подсказок.

Таблица 13.4. *Назначение кнопок специальной панели инструментов диаграммы кооперации*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	Object	Добавляет на диаграмму объект

Таблица 13.4 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Class Instance	Добавляет на диаграмму экземпляр класса
	Object Link	Добавляет на диаграмму связь
	Link To Self	Добавляет на диаграмму рефлексивную связь
	Link Message	Добавляет на связь диаграммы прямое сообщение
	Reverse Link Message	Добавляет на связь диаграммы обратное сообщение
	Data Token	Добавляет на связь диаграммы элемент прямого потока данных
	Reverse Data Token	Добавляет на связь диаграммы элемент обратного потока данных

На специальной панели инструментов по умолчанию присутствует практически все пиктограммы элементов, которые могут быть использованы для построения диаграммы. Напомним, что для разрабатываемого проекта системы управления банкоматом диаграмма кооперации изображена на рис. 6.14. В модели она соответствует варианту использования `Снятие наличных по кредитной карточке` и размещается в представлении вариантов использования (Use Case View).

В общем случае работа с диаграммой кооперации состоит в добавлении или удалении объектов и сообщений, а также заданием и изменением их свойств. При этом изменения, вносимые в диаграмму кооперации, автоматически вносятся в диаграмму последовательности, что можно увидеть в любой момент, активизировав последнюю нажатием клавиши <F5>.

13.3.1. Добавление объекта на диаграмму кооперации

Добавить объект на диаграмму кооперации можно стандартным образом с помощью соответствующей кнопки на специальной панели инструментов. Однако, в случае наличия построенной диаграммы классов (как в нашем случае), более удобным представляется следующий способ. В браузере проекта выделить необходимый класс и, удерживая нажатой левую кнопку мыши, перетащить изображение пиктограммы класса из браузера на свободное место рабочего листа диаграммы.

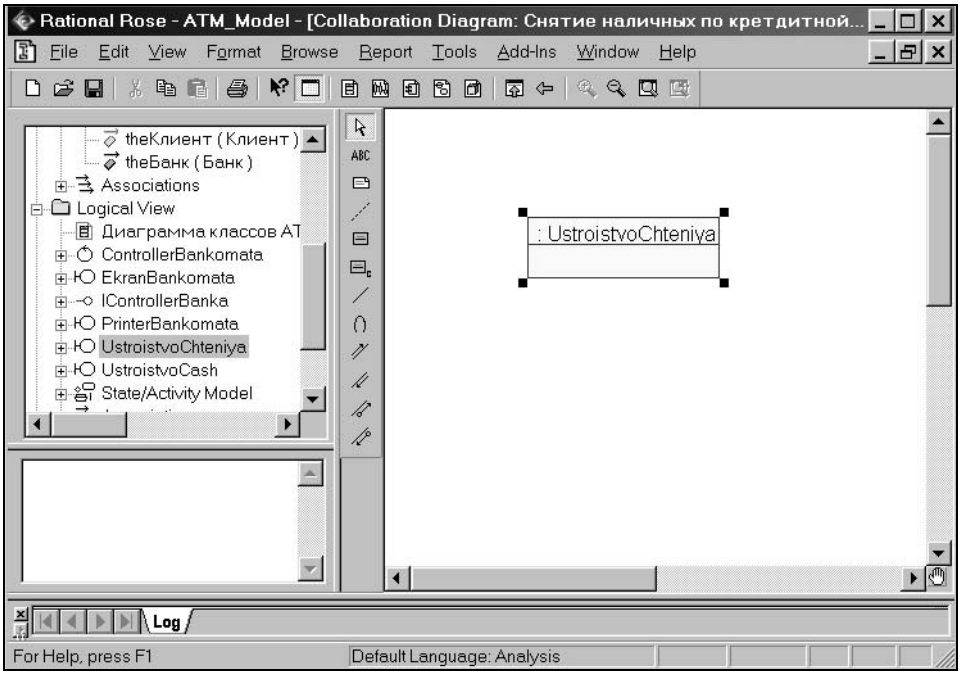


Рис. 13.14. Диаграмма кооперации после добавления на нее анонимного объекта класса `УстройствоЧтения`

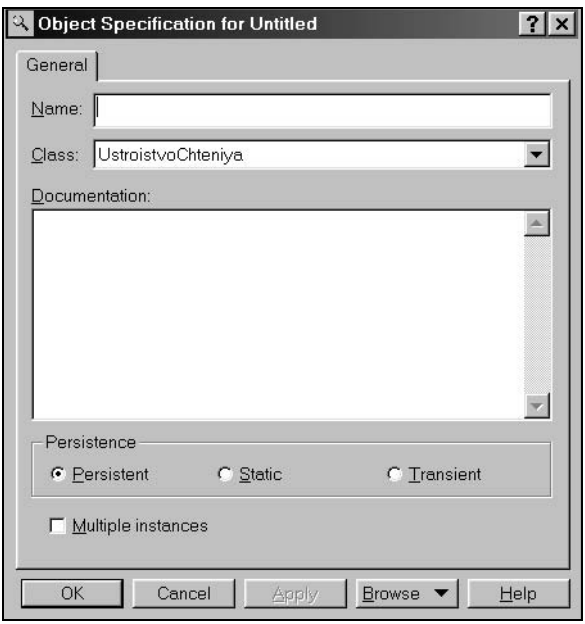


Рис. 13.15. Диалоговое окно настройки свойств объекта

В результате этих действий на диаграмме кооперации появится изображение объекта с именем класса и маркерами изменения его геометрических размеров (рис. 13.14).

По умолчанию каждый добавляемый объект считается анонимным. При необходимости можно задать собственное имя объекта, для чего уже известным способом (например, двойным щелчком на изображении объекта на диаграмме) следует вызвать диалоговое окно свойств объекта (рис. 13.15).

Как видно из рассмотрения этого окна свойств, для объекта выбранного класса можно задавать собственное имя объекта, особенности его реализации и множественность экземпляров.

13.3.2. Добавление связи

Для добавления связи между предварительно размещенными на диаграмме объектами нужно с помощью левой кнопки мыши нажать кнопку с изображением связи на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении одного объекта на диаграмме и отпустить ее на изображении второго объекта. В результате этих действий на диаграмме появится изображение связи, например, соединяющей объект класса `Клиент` (актером) с объектом класса `UstroistvoChteniya` (рис. 13.16).

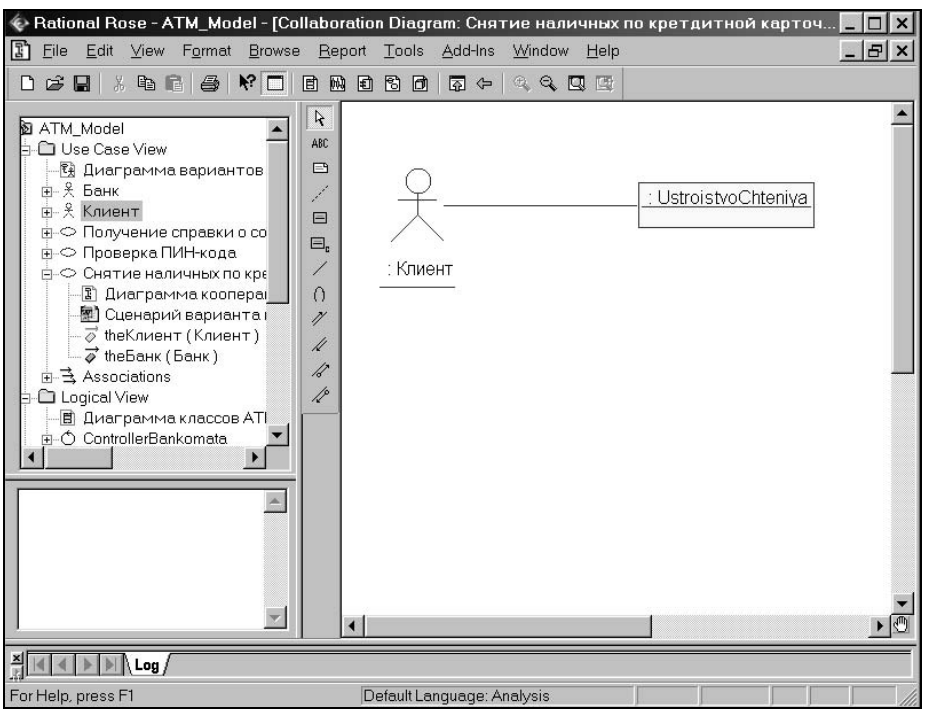


Рис. 13.16. Диаграмма кооперации после добавления связи между объектом класса `Клиент` (актером) и объектом класса `UstroistvoChteniya`

По умолчанию каждая добавляемая связь считается анонимной. При необходимости можно задать имя связи, для чего уже известным способом (например, двойным щелчком на изображении объекта на диаграмме) следует вызвать диалоговое окно свойств связи (рис. 13.17).

Рис. 13.17. Диалоговое окно настройки свойств связи

Кроме имени связи можно также задать: имя ассоциации, видимость соответствующей пары объектов и наличие общих ролей. Однако более важной представляется следующая вкладка **Messages** (Сообщения), служащая для спецификации сообщений, передаваемых между соответствующей парой объектов.

13.3.3. Добавление сообщения

Добавить сообщения на диаграмму кооперации можно несколькими способами. Стандартный способ заключается в использовании кнопки с пиктограммой сообщения на специальной панели инструментов. В этом случае необходимо левой кнопкой мыши нажать кнопку с изображением прямого или обратного сообщения на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении линии связи на диаграмме и отпустить ее. В результате этих действий на диаграмме рядом с линией связи появится изображение стрелки сообщения.

Однако более удобным представляется способ добавления сообщений с помощью диалогового окна свойств связей. Для этого двойным щелчком на линии связи вызывается окно ее свойств и раскрывается вкладка **Messages** (Сообщения). После этого следует выполнить операцию контекстного меню **Insert To** (Вставить в направлении), в результате чего появляется вложенный список с предложением выбрать операцию соответствующего класса для спецификации сообщения. После выбора операции в качестве имени добавляемого сообщения оно добавляется в список сообщений данной связи (рис. 13.18), а рядом с линией связи на диаграмме кооперации появится стрелка с номером и именем этого сообщения.

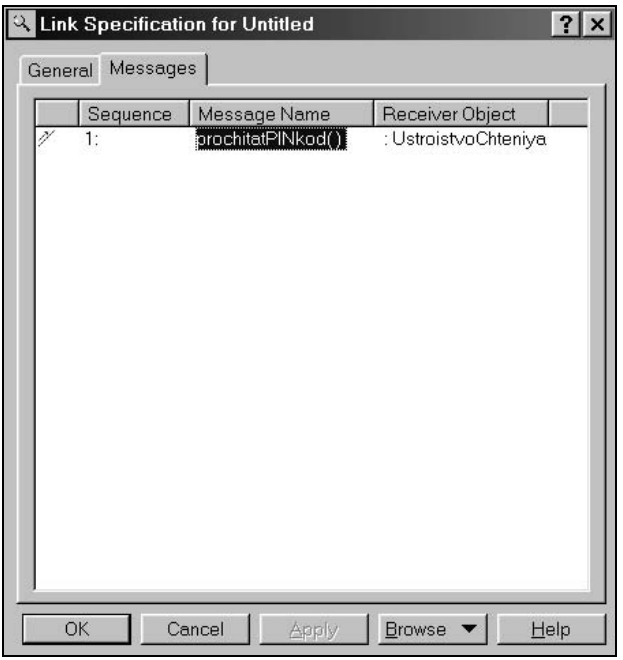


Рис. 13.18. Диалоговое окно настройки свойств сообщения

Кроме имени сообщения можно также задать стереотип синхронизации и частоту передачи. Для этой цели следует воспользоваться диалоговым окном свойств сообщений, которое в свою очередь можно открыть двойным щелчком на имени сообщения в списке рассматриваемой вкладки **Messages** (Сообщения).

Примечание

Следует заметить, что среди свойств сообщений в среде IBM Rational Rose 2002 отсутствует сторожевое условие. Этот факт может несколько огорчить разработчиков, поскольку в общем случае может привести к увеличению количества

диаграмм кооперации в модели проекта. В качестве выхода из данной ситуации можно рекомендовать указать соответствующее сторожевое условие в качестве предусловия соответствующей операции класса.

Для завершения построения диаграммы кооперации рассматриваемого примера следует описанным ранее способом добавить оставшиеся объекты, связи и сообщения. Диаграмма кооперации, описывающая реализацию типичного хода событий варианта использования `Снятие наличных по кредитной карточке` системы управления банкоматом, будет иметь следующий вид (рис. 13.19).

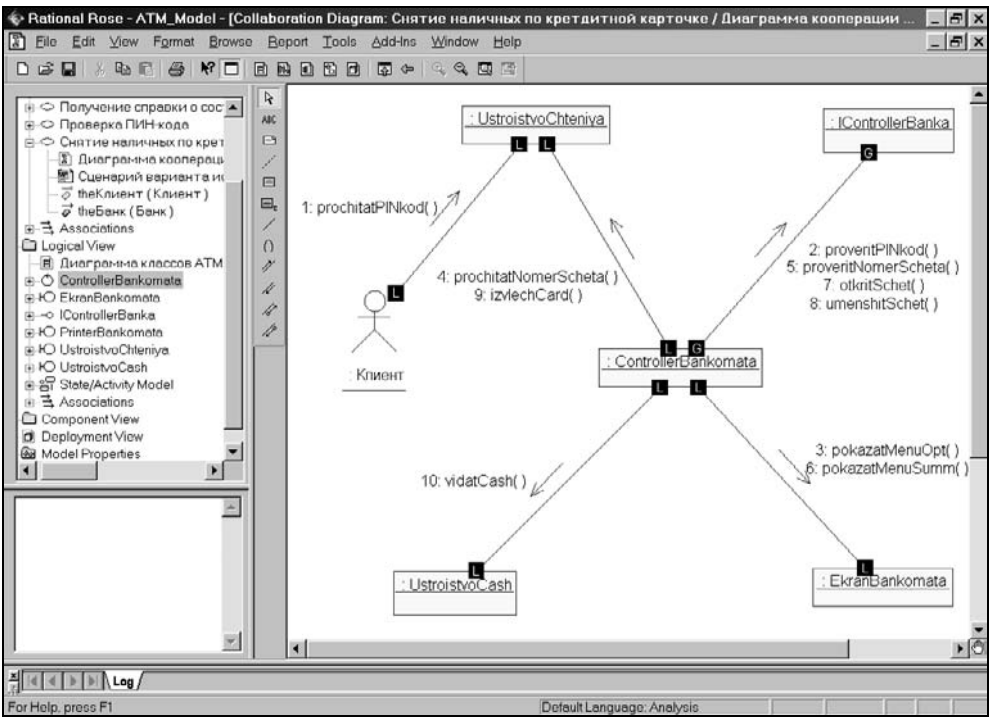


Рис. 13.19. Окончательный вид диаграммы кооперации для варианта использования `Снятие наличных по кредитной карточке`

Следует обратить внимание, что вместо текстовых стереотипов ролей объектов в среде IBM Rational Rose локальность и глобальность визуализируется с помощью небольших маркеров в форме черных квадратов с буквами "`L`" и "`G`" на границе изображения прямоугольника объекта. При необходимости можно изменить порядок следования сообщений и их спецификацию, а также установить дополнительную синхронизацию сообщений и связать с сообщениями примечания. Указанные действия, а также построение других диаграмм кооперации, описывающих реализации других вариантов

использования, предлагается выполнить читателям самостоятельно в качестве упражнения.

13.4. Разработка диаграммы последовательности в среде IBM Rational Rose

Диаграмма последовательности является другой формой визуализации взаимодействия в модели и, как и диаграмма кооперации, оперирует объектами и сообщениями. Общие рекомендации по построению диаграммы последовательности были рассмотрены в *главе 7*. Особенность работы в среде IBM Rational Rose заключается в том, что этот вид канонической диаграммы может быть создан автоматически после построения диаграммы кооперации и нажатия клавиши <F5>. С помощью этой же клавиши осуществляется переключение между диаграммами последовательности и кооперации в модели.

Однако в отдельных случаях бывает удобно начать построение диаграмм взаимодействия с диаграммы последовательности. В этом случае активизировать рабочее окно диаграммы последовательности можно несколькими способами:

- щелкнуть на кнопке с изображением диаграммы взаимодействия на стандартной панели инструментов и выбрать для построения диаграмму последовательности;
- выполнить операцию главного меню **Browse** → **Interaction Diagram** (Обзор → Диаграмма взаимодействия) и выбрать для построения диаграмму последовательности;
- выполнить операцию контекстного меню **New** → **Sequense Diagram** (Новая → Диаграмма последовательности) для логического представления или представления вариантов использования в браузере проекта.

При этом появляется новое окно с чистым рабочим листом диаграммы классов и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необходимых для разработки диаграммы последовательности (табл. 13.5). Назначение отдельных кнопок панели можно узнать из всплывающих подсказок.

Таблица 13.5. *Назначение кнопок специальной панели инструментов диаграммы последовательности*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме

Таблица 13.5 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	Object	Добавляет на диаграмму объект
	Object Message	Добавляет на диаграмму простое сообщение
	Message To Self	Добавляет на диаграмму рефлексивное сообщение
	Return Message	Добавляет на диаграмму сообщение типа возврата из вызова процедуры
	Destruction Marker	Добавляет на диаграмму символ уничтожения объекта
	Procedure Call	Добавляет на диаграмму сообщение типа вызова процедуры (по умолчанию отсутствует)

На специальной панели инструментов по умолчанию присутствует практически все пиктограммы элементов, которые могут быть использованы для построения диаграммы последовательности.

Примечание

Из дополнительных пиктограмм графических элементов, возможно, на специальную панель инструментов следует добавить лишь сообщение типа вызова процедуры (см. последнюю строку табл. 13.4). Относительно пиктограммы асинхронного сообщения в форме полустрелки следует заметить, что в текущей версии языка UML 1.5 этот элемент отсутствует.

Напомним, что для разрабатываемого проекта системы управления банкоматом диаграмма последовательности изображена на рис. 7.11.

13.4.1. Добавление объекта на диаграмму последовательности

Добавить объект на диаграмму последовательности можно как стандартным образом с помощью соответствующей кнопки на специальной панели инструментов, так и более удобным способом — с помощью перетаскивания изображения пиктограммы класса из браузера на свободное место рабочего листа диаграммы.

В результате этих действий на диаграмме кооперации появится изображение объекта с именем класса, маркерами изменения его геометрических размеров и вертикальной пунктирной линией, означающей линию жизни этого объекта (рис. 13.20).

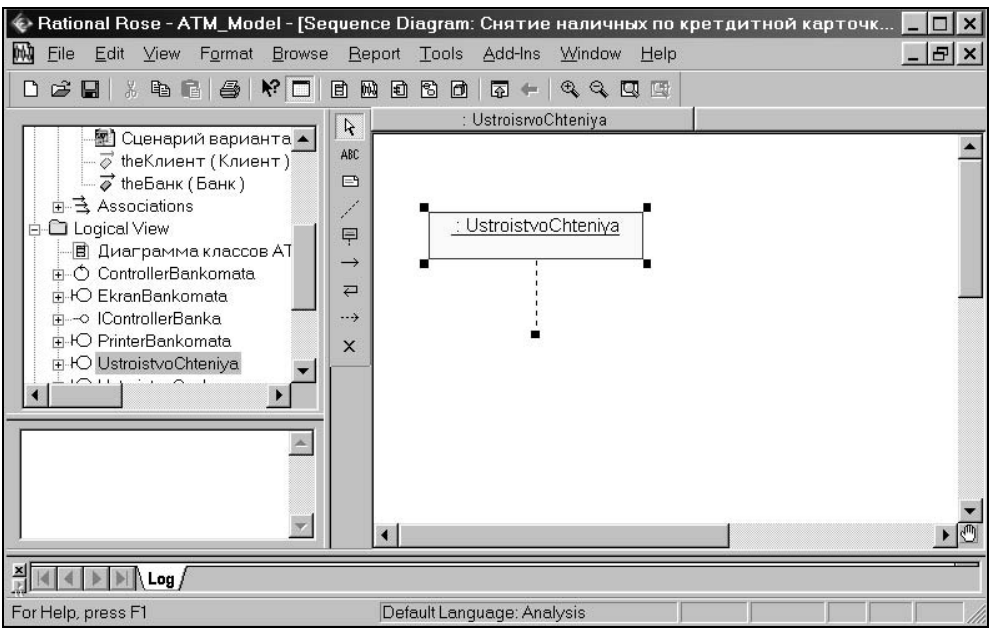


Рис. 13.20. Диаграмма последовательности после добавления анонимного объекта класса `UstroistvoChteniya`

Так же как и для диаграммы кооперации, для диаграммы последовательности каждый добавляемый объект по умолчанию считается анонимным. При необходимости можно задать собственное имя объекта, для чего уже известным способом (например, двойным щелчком на изображении объекта на диаграмме) следует вызвать диалоговое окно свойств объекта, аналогично объектам диаграммы кооперации (рис. 13.15).

13.4.2. Добавление сообщения

Для добавления сообщения между предварительно размещенными на диаграмме объектами нужно с помощью левой кнопки мыши нажать кнопку с изображением сообщения на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении линии жизни одного объекта на диаграмме и отпустить ее на изображении линии жизни второго объекта.

В результате этих действий на диаграмме появится изображение сообщения, например, передаваемого от объекта класса `Клиент` (актером) объекту класса `UstroistvoChteniya` (рис. 13.21).

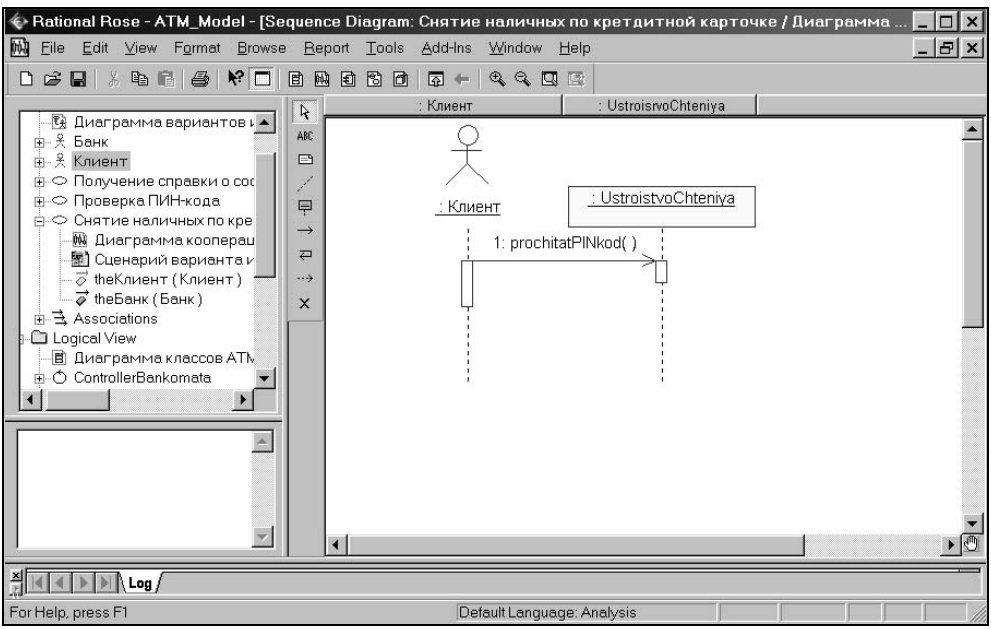


Рис. 13.21. Диаграмма последовательности после добавления сообщения между объектом класса `Клиент` (актером) и объектом класса `UstroistvoChteniya`

Имя сообщения можно выбрать из выпадающего списка операций соответствующего класса-приемника. При необходимости можно задать новую операцию и добавить ее к описанию соответствующего класса.

Построение диаграммы последовательности сводится к добавлению или удалению отдельных объектов и сообщений, а также к их спецификации. Доступ к спецификации свойств этих элементов организован либо через контекстное меню, либо с помощью операции главного меню **Browse → Specification** (Обзор → Спецификация). При добавлении сообщений на диа-

грамму последовательности они получают по умолчанию свой номер в последовательности.

Примечание

По умолчанию нумерация сообщений на диаграмме последовательности может быть отключена. При необходимости показать номера сообщений следует выполнить операцию главного меню **Tools → Options** (Инструменты → Параметры), открыть вкладку **Diagram** (Диаграмма) и выставить отметку выбора строки **Sequence numbering** (Нумерация сообщений на диаграмме последовательности).

Для завершения построения диаграммы последовательности рассматриваемого примера следует описанным ранее способом добавить оставшиеся объекты и сообщения. Диаграмма последовательности реализации типичного хода событий варианта использования `Снятие наличных по кредитной карточке` системы управления банкоматом может иметь следующий вид (рис. 13.22).

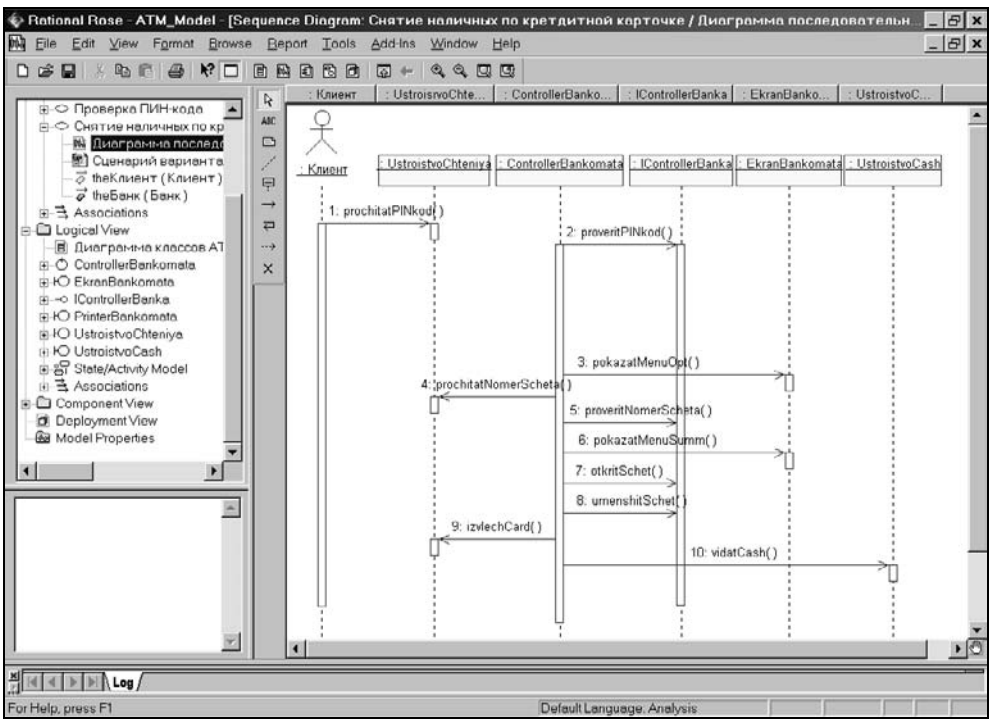


Рис. 13.22. Окончательный вид диаграммы последовательности для варианта использования `Снятие наличных по кредитной карточке`

Если необходимо изменить порядок следования сообщений, то это удобнее сделать на диаграмме последовательности, чем на диаграмме кооперации.

В этом случае достаточно нажать левую кнопку мыши на стрелке соответствующего сообщения и, не отпуская ее, перетащить вертикально вверх или вниз данное сообщение. Дополнительно можно добавлять потоки данных и определять устойчивость объектов на основе активизации соответствующих спецификаций.

Указанные действия, а также построение диаграмм последовательности, описывающих реализации других вариантов использования, предлагается выполнить читателям самостоятельно в качестве упражнения.

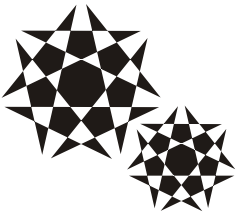
После окончания сеанса работы над проектом выполненную работу необходимо сохранить в файле проекта с расширением MDL. Это можно сделать через меню **File** → **Save** (Файл → Сохранить) или **File** → **Save As** (Файл → Сохранить как). При этом вся информация о проекте, включая диаграммы и спецификации элементов, будет сохранена в одном файле.

Примечание

Вообще говоря, в ходе работы над проектом рекомендуется периодически выполнять сохранение проекта, поскольку невольное зависание операционной системы или средства IBM Rational Rose могут привести к потере всех внесенных изменений в модель со времени ее последнего сохранения. Впрочем, можно установить в настройках опцию автоматического сохранения проекта через определенные промежутки времени. Эта опция по умолчанию отключена.

Для завершения проекта системы управления банкоматом нужно разработать оставшиеся типы канонических диаграмм, из которых необходимой для генерации программного кода является лишь диаграмма компонентов. Тем не менее, из методических соображений разрабатываемая модель будет содержать все типы диаграмм, включая диаграммы состояний, деятельности и развертывания, особенности построения которых в среде IBM Rational Rose рассматриваются в *главе 14*.

Глава 14



Завершение работы над проектом в среде IBM Rational Rose 2002

В настоящей главе описывается процесс завершения разработки проекта в среде IBM Rational Rose 2002. С этой целью рассматриваются особенности построения оставшихся 4-х канонических диаграмм. В заключение приводится информация, необходимая для генерации программного кода на языке ANSI C++ на основе разработанной модели системы управления банкоматом.

Для продолжения работы над проектом системы управления банкоматом следует открыть уже имеющийся файл модели (файл с расширением MDL). Это можно сделать, например, с помощью операции главного меню **File → Open** (Файл → Открыть) либо в результате двойного щелчка на пиктограмме этого файла. В последнем случае, средство IBM Rational Rose 2002 загрузится с существующим проектом со всеми имеющимися в нем диаграммами, спецификациями и документацией, рассмотренными в *главе 13*.

14.1. Разработка диаграммы состояний в среде IBM Rational Rose

Переходя к рассмотрению диаграммы состояний, следует отметить, что этот тип диаграмм может относиться к отдельному классу, варианту использования, пакету или представлению. Общие рекомендации по построению диаграммы состояний были рассмотрены в *главе 8*. Для того чтобы построить диаграмму состояний для класса или пакета, его вначале необходимо создать и специфицировать. Начать построение диаграммы состояний для выбранного класса можно одним из следующих способов:

- щелкнуть на кнопке с изображением диаграммы состояний на стандартной панели инструментов, после следует чего выбрать представление и тип разрабатываемой диаграммы — диаграммы состояний;
- раскрыть логическое представление (Logical View) в браузере проекта, выделить рассматриваемый класс, пакет, вариант использования или

представление, после чего выполнить операцию контекстного меню **New → Statechart Diagram** (Новая → Диаграмма состояний);

- выполнить операцию главного меню **Browse → State Machine Diagram** (Обзор → Диаграмма состояний), после следует чего выбрать представление и тип разрабатываемой диаграммы.

В результате выполнения этих действий появляется новое окно с чистым рабочим листом диаграммы состояний и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необходимых для разработки диаграммы состояний (табл. 14.1). Назначение отдельных кнопок панели можно узнать из всплывающих подсказок.

Таблица 14.1. *Назначение кнопок специальной панели инструментов диаграммы состояний*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	State	Добавляет на диаграмму состояние
	Start State	Добавляет на диаграмму начальное состояние
	End State	Добавляет на диаграмму конечное состояние
	State Transition	Добавляет на диаграмму переход
	Transition to Self	Добавляет на диаграмму рефлексивный переход
	Horizontal Synchronization	Добавляет на диаграмму горизонтально расположенный параллельный переход (по умолчанию отсутствует)
	Vertical Synchronization	Добавляет на диаграмму вертикально расположенный параллельный переход (по умолчанию отсутствует)

Таблица 14.1 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Decision	Добавляет на диаграмму символ принятия решения об альтернативном переходе (по умолчанию отсутствует)

Как видно из этой таблицы, по умолчанию в среде отсутствуют 3 графических элемента из рассмотренных ранее элементов диаграммы состояний. При необходимости их можно добавить на специальную панель диаграммы состояний аналогично способу, рассмотренному ранее в *главе 13*.

Для разрабатываемого проекта системы управления банкоматом диаграмма состояний изображена на рис. 8.15 и строится для системы в целом. В этом случае необходимо в браузере проекта выделить логическое представление (Logical View) и выполнить операцию контекстного меню **New → Statechart Diagram** (Новая → Диаграмма состояний).

14.1.1. Добавление состояния на диаграмму состояний

Для добавления состояния на диаграмму состояний необходимо с помощью левой кнопки мыши нажать кнопку с изображением пиктограммы состояния на специальной панели инструментов, отпустить левую кнопку мыши и щелкнуть левой кнопкой мыши на свободном месте рабочего листа диаграммы. На диаграмме появится изображение состояния с маркерами изменения его геометрических размеров и предложенным по умолчанию именем, которое разработчику следует изменить (рис. 14.1).

Для добавленного состояния можно открыть диалоговое окно его свойств двойным щелчком левой кнопкой мыши на изображении этого элемента на диаграмме. В этом случае активизируется диалоговое окно со специальными вкладками, в поля которых можно занести всю информацию по данному состоянию (рис. 14.2).

При необходимости, можно определить следующие свойства состояний: задать стереотип, специфицировать состояние как историческое, определить внутренние действия на входе и выходе, а также внутреннюю деятельность, которые доступны для редактирования на вкладке **Actions** (Действия). На вкладке **Transitions** (Переходы) можно определять и редактировать переходы, которые входят и выходят из рассматриваемого состояния. Последняя вкладка **Swimlanes** (Дорожки) служит для спецификации дорожек, которые, в контексте языка UML, определяются для диаграммы деятельности.

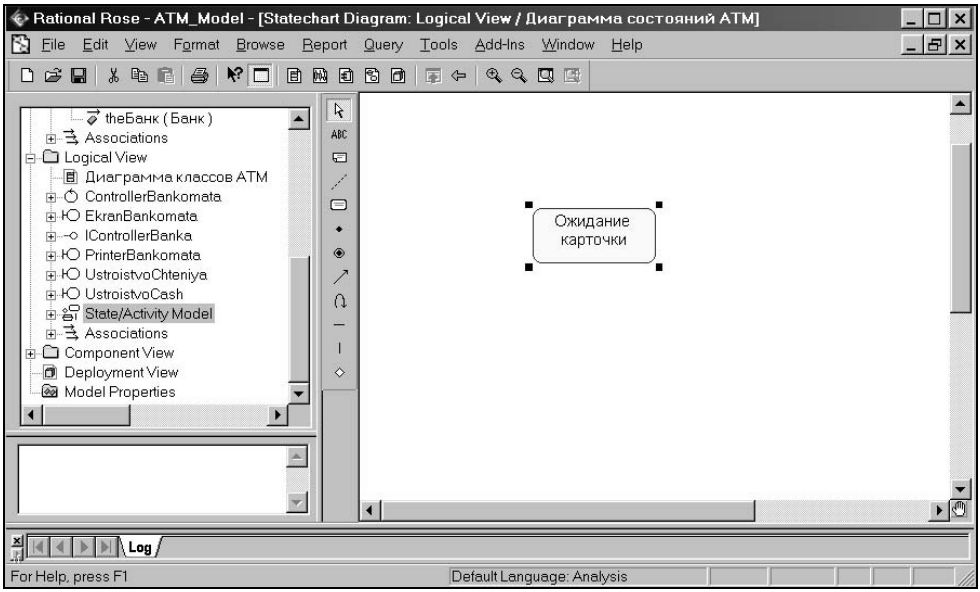


Рис. 14.1. Диаграмма состояний после добавления на нее состояния `Ожидание карточки`

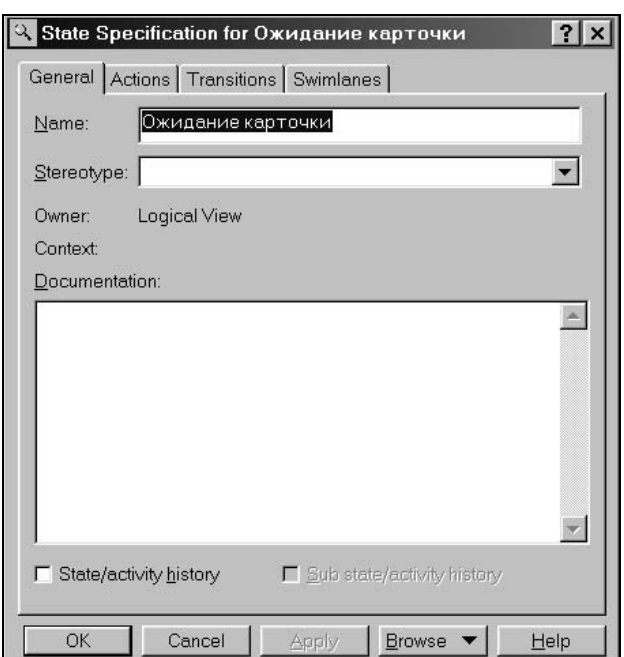


Рис. 14.2. Диалоговое окно настройки свойств состояния

14.1.2. Добавление перехода

Для добавления перехода между двумя состояниями нужно с помощью левой кнопки мыши нажать кнопку с изображением перехода на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении исходного состояния на диаграмме и отпустить ее на изображении целевого состояния. В результате этих действий на диаграмме появится изображение перехода, соединяющего два выбранных состояния (рис. 14.3).

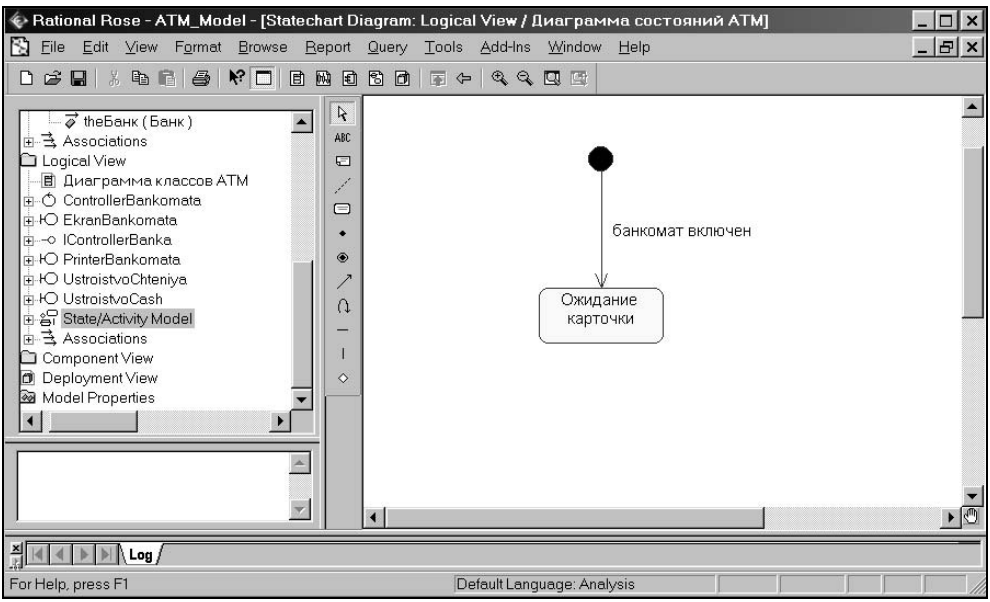


Рис. 14.3. Диаграмма состояний после добавления на нее перехода из начального состояния в состояние `Ожидание карточки`

После добавления перехода на диаграмму состояний можно открыть диалоговое окно его свойств и специфицировать дополнительные свойства, доступные на соответствующих вкладках (рис. 14.4). Следует обратить внимание на две первые строки вкладки **Detail**, которые представляются наиболее важными из свойств перехода. Первая из строк — **Guard Condition**, служит для задания сторожевого условия, которое определяет правило срабатывания перехода. Во второй строке — **Action**, можно специфицировать действие, которое происходит при срабатывании перехода.

При необходимости можно определить сообщение о событии, происходящем при срабатывании перехода, а также визуализировать вложенность состояний и подключить историю отдельных состояний. Для окончательного построения

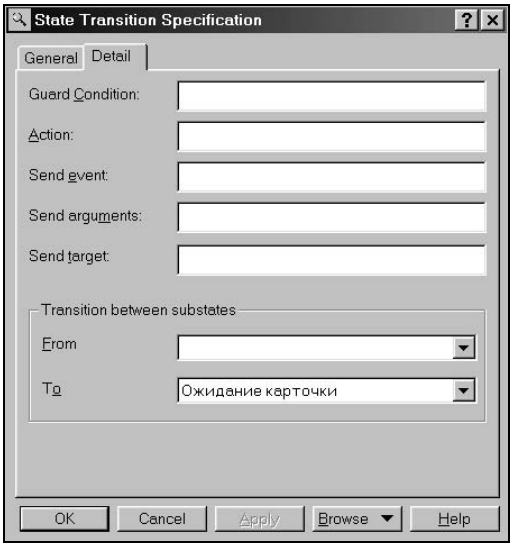


Рис. 14.4. Диалоговое окно настройки свойств перехода, открытое на вкладке **Detail**

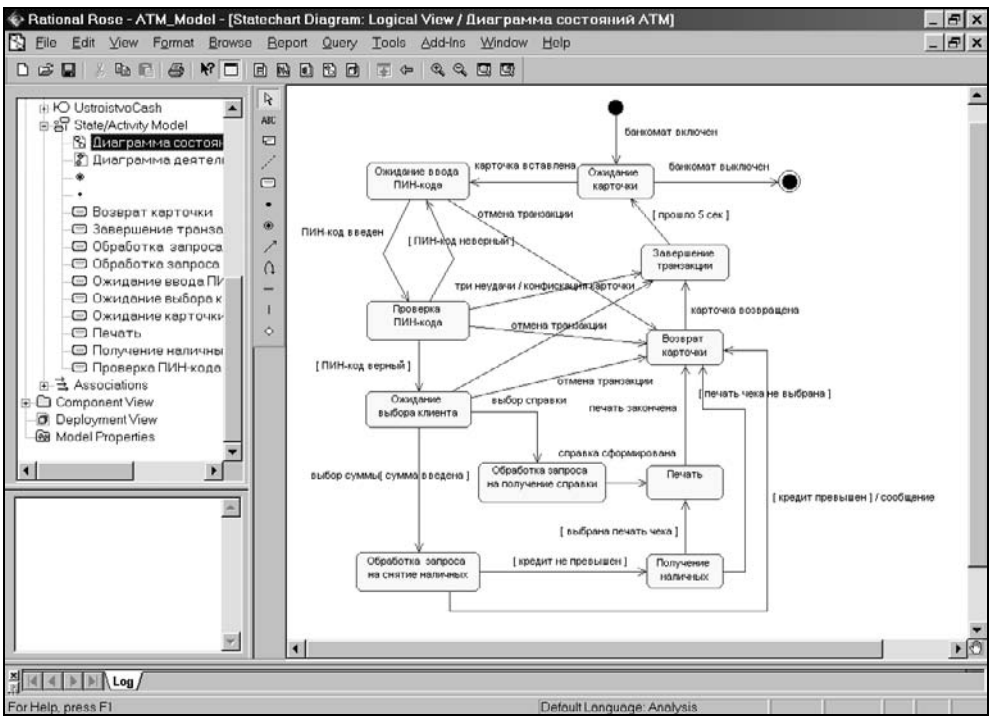


Рис. 14.5. Окончательный вид диаграммы состояний разрабатываемой модели управления банкоматом

диаграммы состояний рассматриваемого примера следует аналогичным образом добавить недостающие состояния и переходы. Построенная таким образом диаграмма состояний будет иметь следующий вид (рис. 14.5).

Следует заметить, что в разрабатываемой модели диаграмма состояний является единственной и описывает поведение системы управления банкоматом в целом. Главное достоинство данной диаграммы состояний — возможность визуализировать условный характер реализации всех вариантов использования в форме изменения отдельных состояний разрабатываемой системы. В то же время, в среде IBM Rational Rose 2002, данная диаграмма не является необходимой для генерации программного кода. Поэтому в случае дублирования информации, представленной на диаграммах взаимодействия (кооперации и последовательности), разработку диаграммы состояний, особенно
в условиях дефицита времени, отпущенного на выполнение проекта, иногда опускают.

14.2. Разработка диаграммы деятельности в среде IBM Rational Rose

Переходя к рассмотрению диаграммы деятельности, следует отметить, что в среде IBM Rational Rose 2002 этот тип диаграмм, так же как и диаграмма состояний, может относиться к отдельному классу, варианту использования, пакету или представлению. Общие рекомендации по построению диаграммы деятельности были рассмотрены в *главе 9*. Для того чтобы построить диаграмму деятельности для класса, пакета или варианта использования, его вначале необходимо создать и специфицировать. После этого выделить на диаграмме или в браузере проекта. Начать построение диаграммы деятельности для выбранного класса, пакета или варианта использования можно одним из следующих способов:

- щелкнуть на кнопке с изображением диаграммы состояний на стандартной панели инструментов, после следует чего выбрать представление и тип разрабатываемой диаграммы — диаграмма деятельности;
- раскрыть логическое представление (Logical View) в браузере проекта, выделить рассматриваемый класс, пакет или представление, после чего выполнить операцию контекстного меню **New** → **Activity Diagram** (Новая → Диаграмма деятельности);
- выполнить операцию главного меню **Browse** → **State Machine Diagram** (Обзор → Диаграмма состояний), после чего следует выбрать представление и тип разрабатываемой диаграммы — диаграмма деятельности.

В результате выполнения этих действий появляется новое окно с чистым рабочим листом диаграммы деятельности и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необхо-

димых для разработки диаграммы деятельности (табл. 14.2). Назначение отдельных кнопок панели можно узнать из всплывающих подсказок.

Таблица 14.2. *Назначение кнопок специальной панели инструментов диаграммы деятельности*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
ABC	**Text Box**	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	State	Добавляет на диаграмму состояние
	Activity	Добавляет на диаграмму деятельность
	Start State	Добавляет на диаграмму начальное состояние
	End State	Добавляет на диаграмму конечное состояние
	State Transition	Добавляет на диаграмму переход
	Transition to Self	Добавляет на диаграмму рефлексивный переход
	Horizontal Synchronization	Добавляет на диаграмму горизонтально расположенный параллельный переход (разделение или слияние)
	Vertical Synchronization	Добавляет на диаграмму вертикально расположенный параллельный переход (разделение или слияние)
	Decision	Добавляет на диаграмму символ принятия решения об альтернативном переходе (ветвление или соединение)
	Swimlane	Добавляет на диаграмму дорожку

Таблица 14.2 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Object	Добавляет на диаграмму объект (по умолчанию отсутствует)
	ObjectFlow	Добавляет на диаграмму стрелку потока объектов (по умолчанию отсутствует)

Как видно из этой таблицы, по умолчанию на панели инструментов отсутствуют только два графических элемента из рассмотренных ранее элементов диаграммы деятельности, а именно — кнопки с пиктограммами объекта и потока объектов. При необходимости их можно добавить на специальную панель диаграммы деятельности стандартным способом.

Для разрабатываемого проекта системы управления банкоматом диаграмма деятельности изображена на рис. 9.11 и описывает вариант использования `Снятие наличных по кредитной карточке`. Для удобства включим эту диаграмму в логическое представление, для чего необходимо в браузере проекта выделить логическое представление (Logical View) и выполнить операцию контекстного меню **New** → **Activity Diagram** (Новая → Диаграмма деятельности).

14.2.1. Добавление деятельности на диаграмму деятельности

Для добавления деятельности на диаграмму деятельности нужно с помощью левой кнопки мыши нажать кнопку с изображением пиктограммы деятельности на специальной панели инструментов, отпустить левую кнопку мыши и щелкнуть левой кнопкой мыши на свободном месте рабочего листа диаграммы. На диаграмме появится изображение деятельности с маркерами изменения его геометрических размеров и предложенным средой именем по умолчанию, которое разработчику следует изменить (рис. 14.6).

После добавления деятельности на диаграмму состояний можно открыть диалоговое окно ее свойств и специфицировать дополнительные свойства, доступные на соответствующих вкладках, аналогично свойствам состояний (см. рис. 14.2). На вкладке **Transitions** (Переходы) можно определять и редактировать переходы, которые входят и выходят из рассматриваемой деятельности. Последняя вкладка **Swimlanes** (Дорожки) служит для спецификации дорожки, на которую помещается рассматриваемая деятельность.

Примечание

Хотя среда IBM Rational Rose 2002 позволяет определить свойства деятельности, доступные на вкладке **Actions** (Действия), следует помнить, что внутренние

действия являются свойствами общего состояния, а внутренняя деятельность служит именем собственно деятельности, помещаемой на диаграмму деятельности. Поэтому для деятельности во избежание недоразумений лучше оставить эту вкладку пустой.

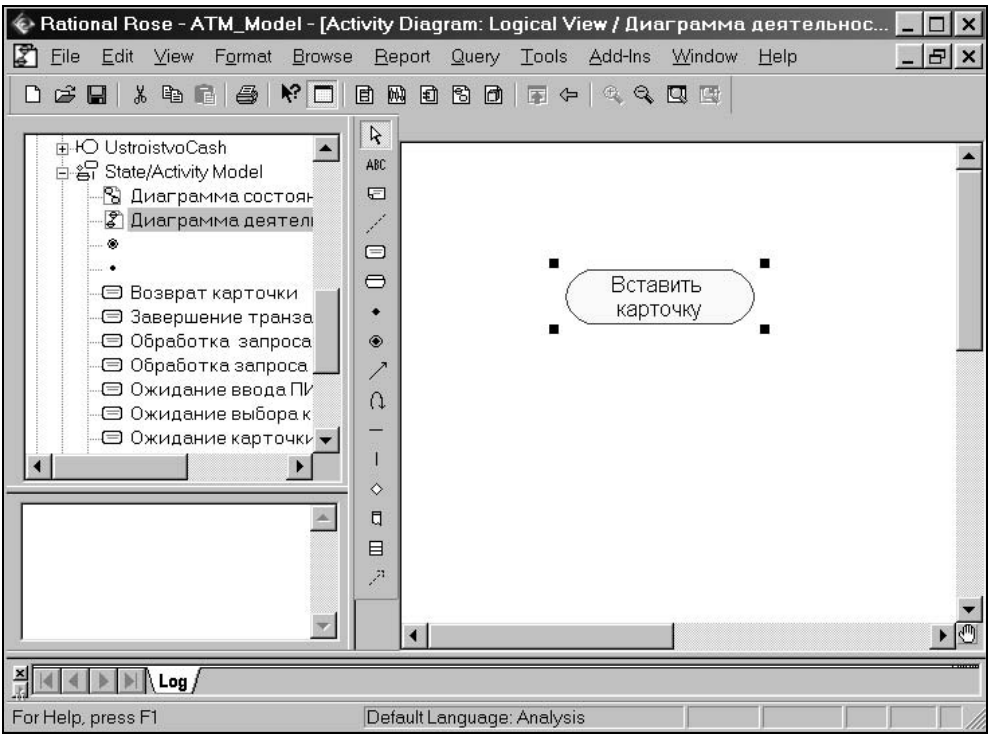


Рис. 14.6. Диаграмма состояний после добавления на нее деятельности `Вставить карточку`

14.2.2. Добавление перехода

Добавление перехода на диаграмму деятельности полностью аналогично диаграмме состояний. А именно, для добавления перехода между двумя деятельностями нужно с помощью левой кнопки мыши нажать кнопку с изображением перехода на специальной панели инструментов, отпустить левую кнопку мыши, щелкнуть левой кнопкой мыши на изображении исходной деятельности на диаграмме и отпустить ее на изображении целевой деятельности. В результате этих действий на диаграмме появится изображение перехода, соединяющего две выбранных деятельности. Если в качестве одной из деятельностей является символ ветвления или соединения, то порядок добавления перехода сохраняется прежним.

После добавления перехода на диаграмму деятельности становятся доступными его свойства в соответствующем диалоговом окне, полностью аналогичном рассмотренному ранее (см. рис. 14.3). При этом следует помнить, что все переходы на диаграмме деятельности являются нетриггерными, т. е. не имеют имен событий. Но все переходы, выходящие из символов ветвления (решения), должны иметь сторожевые условия, которые специфицируются на вкладке **Detail** диалогового окна свойств перехода.

Для окончательного построения диаграммы деятельности рассматриваемого примера следует добавить недостающие деятельности и переходы. Построенная таким образом диаграмма деятельности будет иметь следующий вид (рис. 14.7).

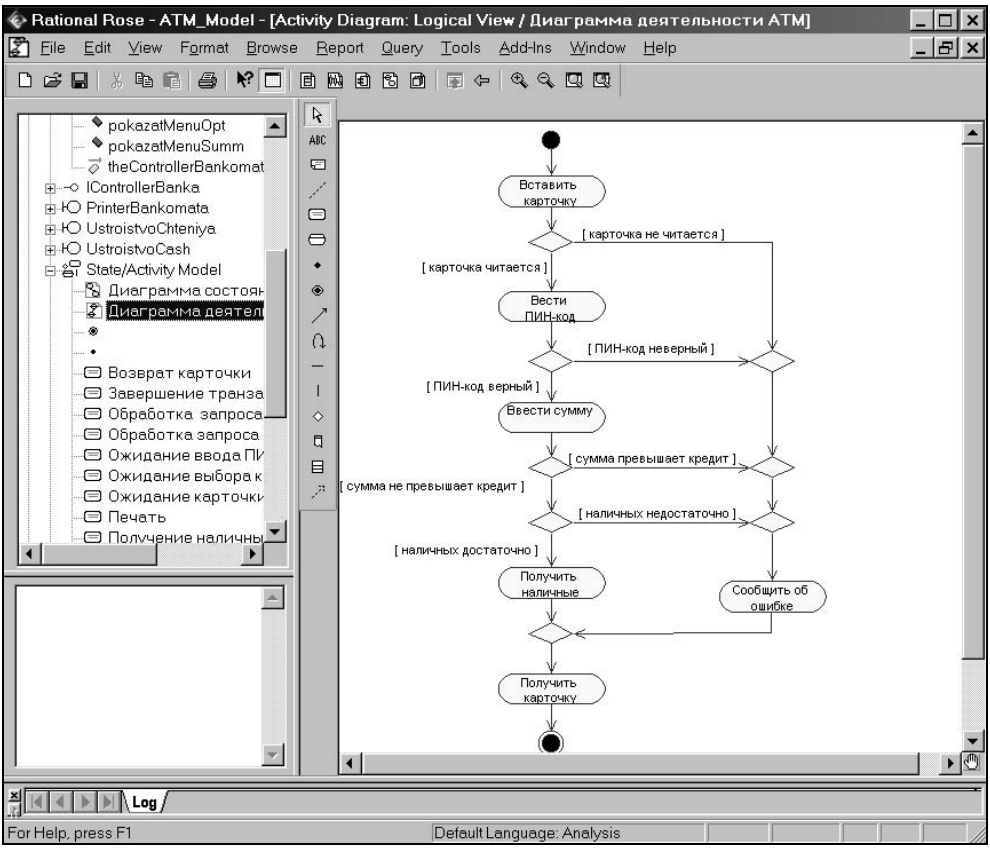


Рис. 14.7. Окончательный вид диаграммы деятельности разрабатываемой модели управления банкоматом

Следует заметить, что в разрабатываемой модели диаграмма деятельности не является единственной, поскольку описывает лишь поведение системы

управления банкоматом при реализации варианта использования `Снятие наличных по кредитной карточке`. Дополнить данную диаграмму информацией по реализации варианта использования `Получение справки о состоянии счета` читателям предлагается самостоятельно в качестве упражнения.

Примечание

В рамках рассматриваемой модели построение диаграммы деятельности для варианта использования `Проверка ПИН-кода`, реализуемого отдельными операциями, может потребовать знания технических аспектов физической реализации кредитной карточки. Именно по этой причине построение соответствующей диаграммы деятельности сопряжено с трудностями концептуального характера и поэтому здесь не приводится.

В общем случае следует помнить, что в среде IBM Rational Rose 2002 диаграмма деятельности не является необходимой для генерации программного кода. Поэтому разработку диаграмм этого типа, особенно в условиях дефицита времени, отпущенного на выполнение проекта, иногда опускают. В то же время следует отметить, что в проектах реинжиниринга и документирования бизнес-процессов диаграмма деятельности является основным средством визуализации бизнес-процессов в контексте языка UML.

14.3. Разработка диаграммы компонентов в среде IBM Rational Rose

Диаграмма компонентов служит частью физического представления модели, играет важную роль в процессе ООАП и является необходимой для генерации программного кода. Общие рекомендации по построению диаграммы компонентов были рассмотрены в *главе 10*. Для диаграмм компонентов в модели предназначено отдельное представление компонентов (Component View). Активизация диаграммы компонентов может быть выполнена одним из следующих способов:

- щелкнуть на кнопке с изображением диаграммы компонентов на стандартной панели инструментов;
- раскрыть компонентное представление в браузере (Component View) и дважды щелкнуть на пиктограмме **Main** (Главная);
- через пункт меню **Browse → Component Diagram** (Обзор → Диаграмма компонентов).

В результате выполнения этих действий появляется новое окно с чистым рабочим листом диаграммы компонентов и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необходимых для разработки диаграммы компонентов (табл. 14.3).

Таблица 14.3. *Назначение кнопок специальной панели инструментов диаграммы компонентов*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	Component	Добавляет на диаграмму компонент
	Package	Добавляет на диаграмму пакет
	Dependency	Добавляет на диаграмму отношение зависимости
	Subprogram Specification	Добавляет на диаграмму спецификацию подпрограммы
	Subprogram Body	Добавляет на диаграмму тело подпрограммы
	Main Program	Добавляет на диаграмму главную программу
	Package Specification	Добавляет на диаграмму спецификацию пакета
	Package Body	Добавляет на диаграмму тело пакета
	Task Specification	Добавляет на диаграмму спецификацию задачи
	Task Body	Добавляет на диаграмму тело задачи
	Generic Subprogram	Добавляет на диаграмму типовую подпрограммы (по умолчанию отсутствует)
	Generic Package	Добавляет на диаграмму типовой пакет (по умолчанию отсутствует)
	Database	Добавляет на диаграмму базу данных (по умолчанию отсутствует)

Как видно из этой таблицы, по умолчанию на панели инструментов отсутствуют только три графических элемента из рассмотренных ранее элементов диаграммы компонентов, а именно — кнопки с пиктограммами типовых подпрограммы и пакета, а также базы данных. При необходимости их можно добавить на специальную панель диаграммы деятельности стандартным способом.

Поскольку средство IBM Rational Rose 2002 не поддерживает рассмотренные в *главе 10* графические стереотипы и предлагает целый ряд собственных стереотипов, эти обстоятельства приводят к существенной модификации предложенной ранее диаграммы компонентов системы управления банкоматом (см. рис. 10.9).

14.3.1. Добавление компонента на диаграмму компонентов

Добавление и удаление компонентов на диаграмму компонентов выполняется стандартным образом с помощью кнопок с изображением пиктограммы требуемого компонента. После добавления компонента на диаграмму (рис. 14.8) следует определить его имя. С помощью соответствующих маркеров можно изменить также геометрические размеры компонента.

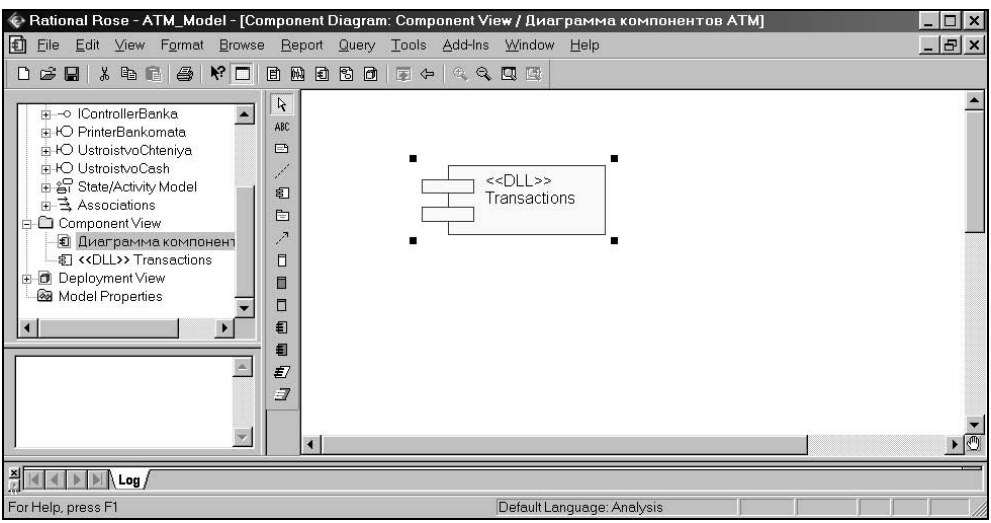


Рис. 14.8. Диаграмма компонентов после добавления компонента `Transactions` со стереотипом библиотеки динамической компоновки

Для каждого компонента можно определить различные свойства, такие как: стереотип, язык программирования, декларации, реализуемые классы. Спецификация свойств этих компонентов осуществляется с помощью диалогового окна свойств (рис. 14.9).

Рис. 14.9. Диалоговое окно настройки свойств компонента `Transactions`

В частности для компонента `Transactions` следует выбрать стереотип <<DLL>> из предлагаемого вложенного списка, поскольку применительно к разрабатываемой модели предполагается реализация этого компонента в форме библиотеки динамической компоновки.

Примечание

Следует отметить, что по умолчанию для всех добавляемых на диаграмму компонентов в качестве языка реализации выбран язык анализа, который следует изменить на тот язык программирования, который предполагается использовать для написания программного кода. Средство IBM Rational Rose 2002 поддерживает возможность использования различных языков программирования для реализации отдельных компонентов модели. Особенности языковой реализации модели рассматриваются далее в *подразделе 14.5*.

14.3.2. Добавление отношения зависимости

Добавление отношения зависимости на диаграмму компонентов аналогично добавлению соответствующего отношения на диаграмму вариантов использования. В результате на диаграмме компонентов появится изображение отношения зависимости между двумя компонентами (рис. 14.10).

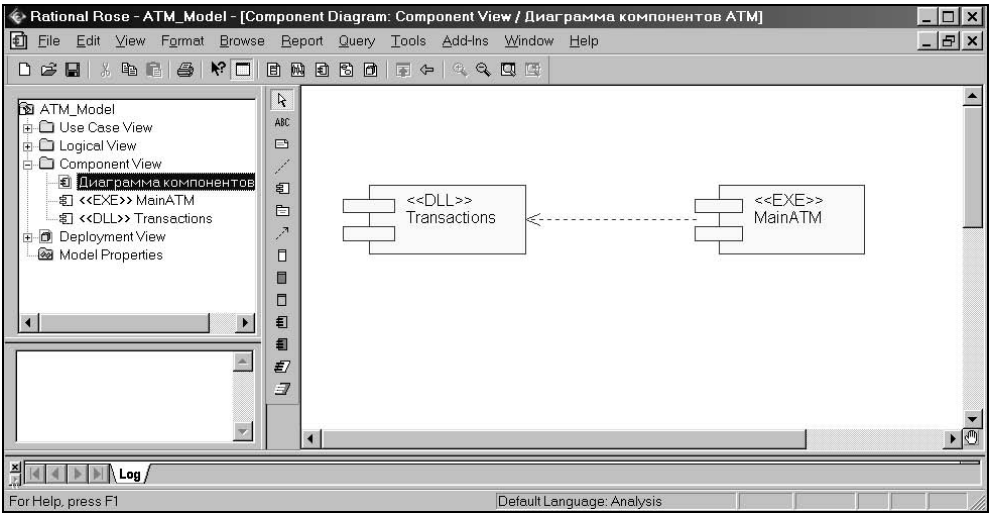


Рис. 14.10. Диаграмма компонентов после добавления отношения зависимости между компонентами `Transactions` и `MainATM`

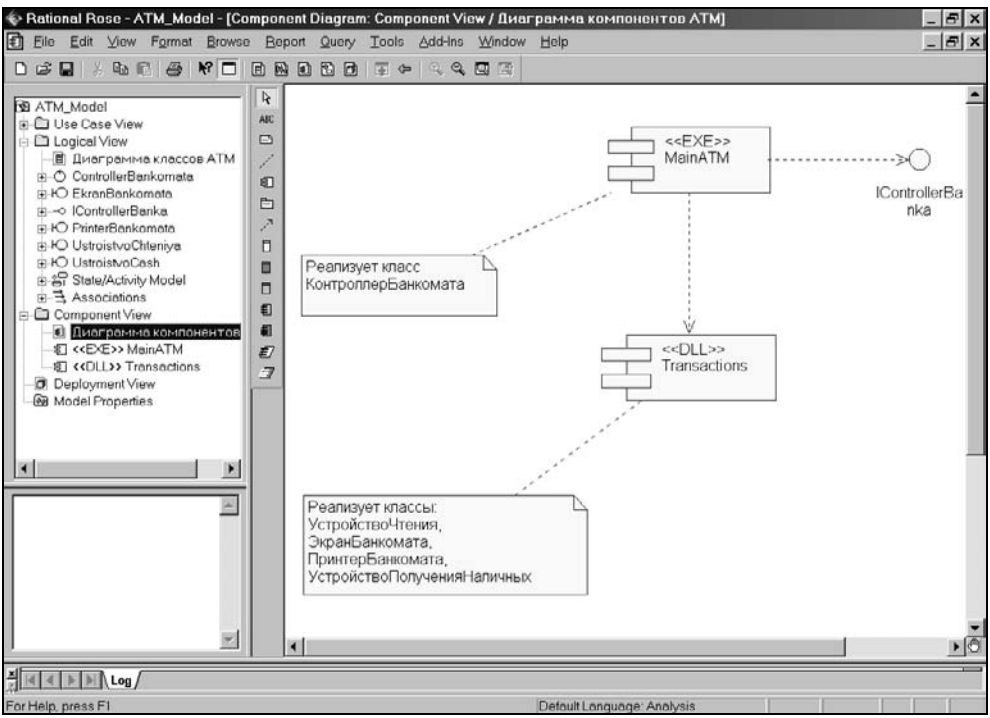


Рис. 14.11. Окончательный вид диаграммы компонентов разрабатываемой модели управления банкоматом

Для окончательного построения диаграммы компонентов рассматриваемого примера следует добавить недостающие компоненты, отношения зависимости и примечания. Для добавления на диаграмму компонентов интерфейса следует выделить его в браузере проекта и перетащить на рабочий лист диаграммы. Примечания не являются необходимыми для данной диаграммы, но значительно повышают наглядность визуализации модели. Построенная таким образом диаграмма компонентов будет иметь следующий вид (рис. 14.11).

При работе с диаграммой компонентов можно создавать пакеты и размещать в них компоненты, использовать специальные графические стереотипы для отдельных разновидностей компонентов, изменять их спецификацию и отношения зависимости между различными элементами диаграммы. Выполнить эти действия предлагается читателям самостоятельно в качестве упражнения.

14.4. Разработка диаграммы развертывания в среде IBM Rational Rose

Диаграмма развертывания является второй составной частью физического представления модели и разрабатывается, как правило, для территориально распределенных систем. Общие рекомендации по построению диаграммы развертывания были рассмотрены в *главе 11*. Активизация диаграммы развертывания может быть выполнена одним из следующих способов:

- щелкнуть на кнопке с изображением диаграммы развертывания на стандартной панели инструментов;
- дважды щелкнуть на пиктограмме представления развертывания (Deployment View) в браузере проекта;
- выполнить операцию главного меню **Browse → Deployment Diagram** (Обзор → Диаграмма развертывания).

В результате выполнения этих действий появляется новое окно с чистым рабочим листом диаграммы развертывания и специальная панель инструментов, содержащая кнопки с изображением графических примитивов, необходимых для разработки диаграммы развертывания (табл. 14.4).

Таблица 14.4. *Назначение кнопок специальной панели инструментов диаграммы развертывания*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Selection Tool	Превращает изображение курсора в форму стрелки для последующего выделения элементов на диаграмме

Таблица 14.4 *(окончание)*

Графическое изображение	Всплывающая подсказка	Назначение кнопки
	Text Box	Добавляет на диаграмму текстовую область
	Note	Добавляет на диаграмму примечание
	Anchor Note to Item	Добавляет на диаграмму связь примечания с соответствующим графическим элементом диаграммы
	Processor	Добавляет на диаграмму процессор
	Connection	Добавляет на диаграмму отношение соединения
	Device	Добавляет на диаграмму устройство

Как видно из этой таблицы, по умолчанию на панели инструментов присутствуют все графические элементы из рассмотренных ранее элементов диаграммы развертывания, поэтому изменять специальную панель нет необходимости. Работа с диаграммой развертывания состоит в создании процессоров и устройств, их спецификации, установлении связей между ними, а также добавлении и спецификации процессов.

14.4.1. Добавление узла на диаграмму развертывания

Добавление узла на диаграмму развертывания выполняется стандартным образом с помощью кнопок с изображением пиктограммы требуемого узла — процессора или устройства. Напомним, что в среде IBM Rational Rose 2002 под процессором понимается ресурсоемкий узел, а под устройством — нересурсоемкий узел.

После добавления узла на диаграмму (рис. 14.12) следует определить его имя, а с помощью соответствующих маркеров можно изменить его геометрические размеры.

Для каждого процессора можно определить различные свойства, такие как стереотип, характеристику, процессы и их приоритет. Спецификация этих свойств процессора осуществляется с помощью диалогового окна свойств (рис. 14.13). При этом для задания стереотипа следует ввести его текст без угловых кавычек в строку с именем **Stereotype**.

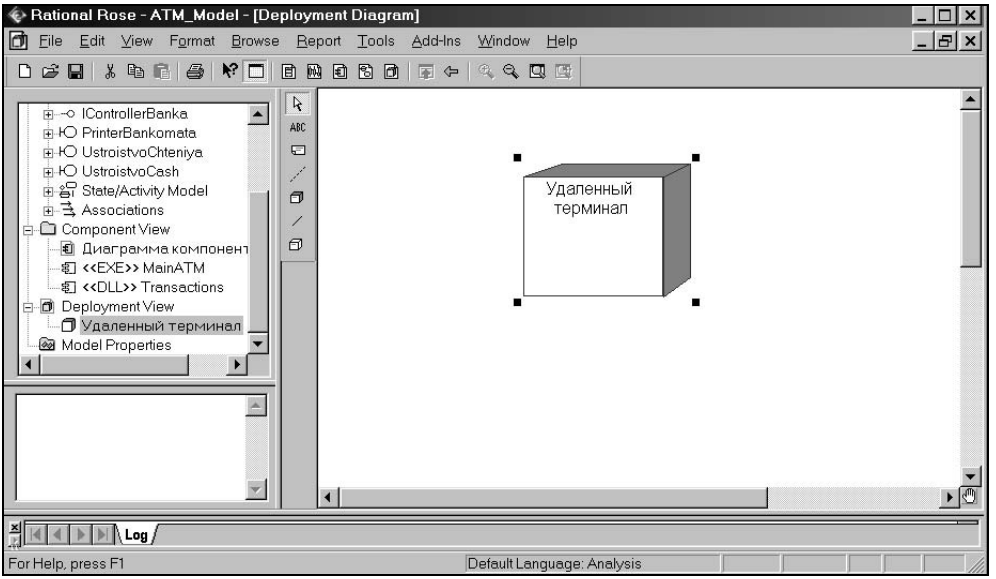


Рис. 14.12. Диаграмма развертывания после добавления узла `Удаленный терминал`

Рис. 14.13. Диалоговое окно настройки свойств узла `Удаленный терминал`

Для устройства набор редактируемых свойств меньше, поскольку для него с помощью соответствующего окна свойств можно определить только стереотип и характеристику. Этот факт согласуется с определением устройства как узла, на котором отсутствует процессор.

14.4.2. Добавление соединения

Добавление соединения на диаграмму развертывания осуществляется способом, аналогичным ранее рассмотренным добавлениям стрелок различных отношений. В результате выбора соединения на диаграмме развертывания появится изображение линии соединения между двумя узлами (рис. 14.14).

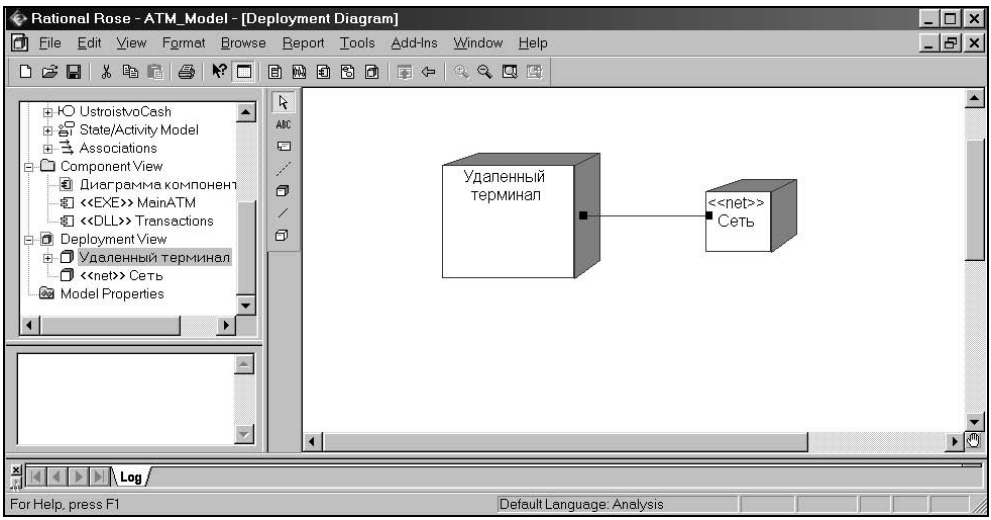


Рис. 14.14. Диаграмма развертывания после добавления соединения между узлами `Удаленный терминал` и `Сеть`

Для окончательного построения диаграммы развертывания рассматриваемого примера следует добавить недостающие узлы и соединения между ними. Построенная таким образом диаграмма компонентов будет иметь следующий вид (рис. 14.15).

Нужно отметить, что среда IBM Rational Rose 2002 не поддерживает графическое изображение размещения внутри узлов развертываемых на них компонентов. Именно по этой причине окончательный вариант диаграммы развертывания отличается от рассмотренной в *главе 11* диаграммы развертывания (см. рис. 11.8). Построением диаграммы развертывания завершается разработка модели в нотации языка UML для системы управления банкоматом.

Дальнейшая работа с моделью зависит от целей выполнения проекта. Если проект не предполагает программную реализацию, то можно ограничиться формированием проектной документации. С этой целью следует выполнить

операцию главного меню **Report → SoDA Report** (Отчет → Отчет с помощью SoDA), в результате чего будет открыто диалоговое окно свойств для выбора шаблоном генерации отчета. После выбора шаблонов будет автоматически сгенерирован отчет о разрабатываемой модели в формате MS Word с использованием специального средства IBM Rational SoDA (если оно было установлено в системе при инсталляции IBM Rational Rose).

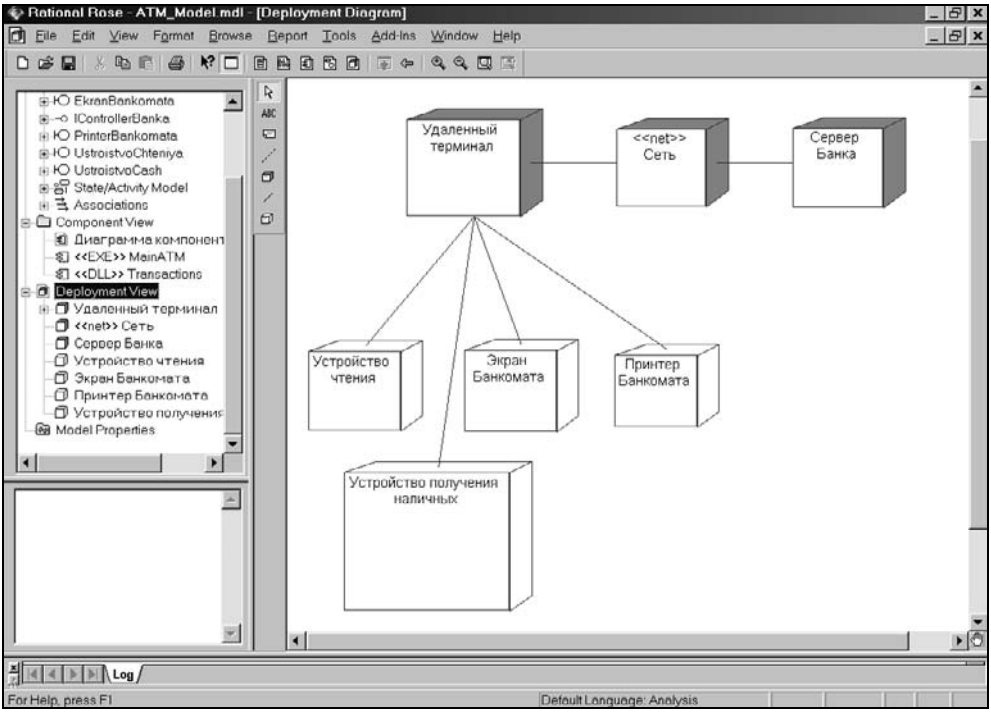


Рис. 14.15. Окончательный вид диаграммы развертывания разрабатываемой модели управления банкоматом

Если проект предполагает программную реализацию, то следует познакомиться с особенностями генерации программного кода в среде IBM Rational Rose 2002, на основе разработанной модели.

14.5. Генерации программного кода в среде IBM Rational Rose

Одним из наиболее мощных свойств является возможность генерации программного кода после построения модели. Для этой цели присутствует большой выбор языков программирования и схем баз данных, однако возможность генерации текста программы на том или ином языке программирования зависит от установленной версии IBM Rational Rose 2002.

Общая последовательность действий, которые необходимо выполнить для генерации программного кода в среде IBM Rational Rose 2002, состоит из нескольких этапов.

1. Проверка модели независимо от выбора языка генерации кода.
2. Создание компонентов для реализации классов.
3. Отображение классов на компоненты.
4. Выбор языка программирования для генерации текста программного кода.
5. Установка свойств генерации программного кода.
6. Выбор класса, компонента или пакета.
7. Генерация программного кода.

Особенности выполнения каждого из этапов могут изменяться в зависимости от выбора языка программирования или схемы базы данных. В среде IBM Rational Rose 2002 предусмотрено задание достаточно большого числа свойств, характеризующих как отдельные классы, так и проект в целом. Для определенности выберем в качестве языка реализации проекта язык программирования ANSI C++, который не требует изучения дополнительных пакетов библиотек и поставляется практически во всех конфигурациях IBM Rational Rose 2002. Рассмотрим особенности выполнения каждого из этапов для выбранного языка программирования ANSI C++.

14.5.1. Проверка модели независимо от выбора языка генерации кода

В общем случае проверка модели может выполняться на любом этапе работы над проектом. Однако после завершения разработки графических диаграмм она является обязательной, поскольку позволяет выявить целый ряд ошибок разработчика. К числу таких ошибок и предупреждений относятся, например, неиспользуемые ассоциации и классы, оставшиеся после удаления отдельных графических элементов с диаграмм, а также операции, не являющиеся именами соответствующих сообщений.

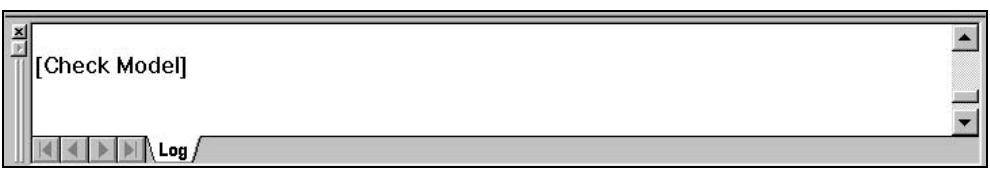


Рис. 14.16. Вид журнала при отсутствии ошибок в результате проверки модели

Для проверки модели следует выполнить операцию главного меню **Tools** → **Check Model** (Инструменты → Проверить модель). Результаты проверки

разработанной модель на наличие ошибок отображаются в окне журнала. Прежде чем приступить, собственно, к генерации текста программного кода разработчику следует добиться устранения всех ошибок и предупреждений, о чем должно свидетельствовать чистое окно журнала (рис. 14.16).

14.5.2. Создание компонентов для реализации классов

По существу данный этап выполняется в ходе разработки диаграммы компонентов. Однако среда IBM Rational Rose 2002 позволяет генерировать текст программного кода на языке ANSI C++ для каждого класса модели без предварительного построения диаграммы компонентов. Поскольку применительно к разрабатываемому проекту данный этап уже выполнен, нет необходимости повторно описывать особенности построения диаграммы компонентов. При желании можно обратиться к материалу *подраздела 14.3*.

14.5.3. Отображение классов на компоненты

Для отображения классов на компоненты можно воспользоваться окном спецификации свойств компонента, открытого на вкладке **Realizes** (Реализации).

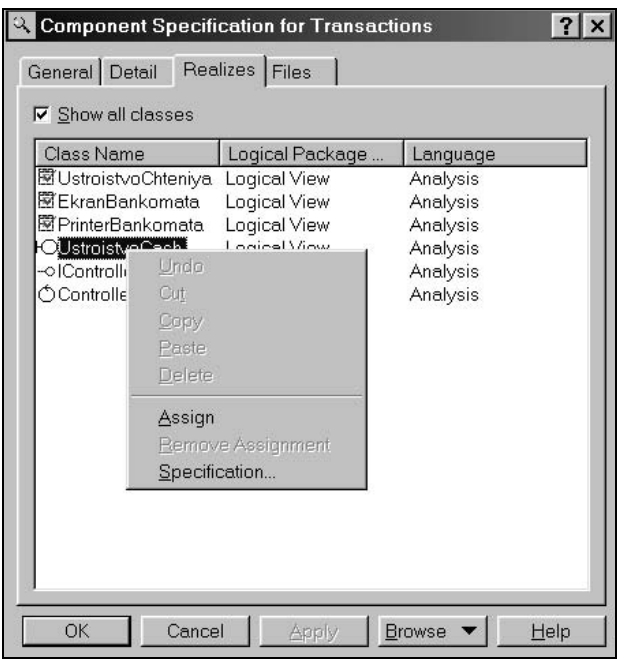


Рис. 14.17. Диалоговое окно настройки свойств реализации классов в компоненте `Transactions`

Для включения реализации класса в данный компонент следует выделить требуемый класс на этой вкладке и в контекстном меню (рис. 14.17), вызываемом по щелчку правой кнопки мыши, выполнить операцию **Assign** (Назначить).

После назначения выбранного класса для реализации, в данном компоненте появится отметка слева в строке с именем этого класса. Подобная операция должна быть выполнена для всех классов модели, которые предполагается реализовывать на выбранном языке программирования.

Примечание

Возможно, некоторым читателям более удобным покажется другой способ установления реализации классов на компоненте, для чего можно выделить класс в браузере и перетащить его на нужный компонент диаграммы компонентов.

14.5.4. Выбор языка программирования для генерации текста программного кода

Для выбора языка ANSI C++ в качестве языка реализации модели следует выполнить операцию главного меню **Tools** → **Options** (Инструменты → Параметры), в результате чего будет вызвано диалоговое окно настройки параметров модели. Далее, на вкладке **Notation** (Нотация), в строке **Default Language** (Язык по умолчанию) из вложенного списка следует выбрать язык — ANSI C++.

Примечание

Если по какой-то причине языка ANSI C++ не оказалось во вложенном списке, то следует убедиться в том, что этот язык установлен в качестве расширения IBM Rational Rose 2002. Для этого следует открыть окно установленных расширений (рис. 12.17), выполнив операцию главного меню **Add-Ins** (Расширения), и убедиться в том, что выставлена отметка в строке с именем языка ANSI C++. Если ее нет, то ее следует добавить, после чего появится группа доступных операций **ANSI C++** главного меню **Tools**.

После выбора языка программирования по умолчанию следует изменить язык реализации каждого из компонентов модели. С этой целью следует изменить язык анализа на вкладке **General** (Общие), в строке **Language** (Язык), для чего из вложенного списка следует выбрать язык — ANSI C++ (см. рис. 14.9).

14.5.5. Установка свойств генерации программного кода

Редактирование общих свойств генерации программного кода возможно в специальном диалоговом окне, которое может быть открыто в результате выполнения операции главного меню **Tools → ANSI C++ → Open ANSI C++ Specification** (Инструменты → Язык ANSI C++ → Открыть спецификацию языка ANSI C++).

Дополнительные свойства генерации программного кода отдельного класса можно специфицировать в диалоговом окне, которое может быть открыто в результате выполнения операции контекстного меню **ANSI C++ → Class Customization** (Язык ANSI C++ → Настройка свойств класса). При этом соответствующий класс должен быть выделен в браузере проекта.

Применительно к генерации программного кода на языке ANSI C++, можно оставить без изменения значения свойств, предлагаемых средой IBM Rational Rose 2002, по умолчанию.

14.5.6. Выбор класса, компонента или пакета для генерации программного кода

Выбор класса, компонента или пакета для генерации программного кода по сути означает выделение соответствующего элемента модели в браузере проекта. Применительно к рассматриваемой модели системы управления банкоматом для генерации программного кода на языке ANSI C++ выберем компонент с именем `Transactions`.

14.5.7. Генерация программного кода

Генерация программного кода в среде IBM Rational Rose 2002 возможна для отдельного класса, пакета или компонента. Для этого нужный элемент модели предварительно следует выделить в браузере проекта и выполнить операцию контекстного меню **ANSI C++ → Generate Code** (Язык ANSI C++ → Генерировать код). В результате этого будет открыто диалоговое окно с предложением выбрать папку для размещения файлов с текстом программного кода на выбранном языке программирования. После выбора соответствующей папки и закрытия этого окна средство IBM Rational Rose 2002 выполняет генерацию кода.

Для просмотра и редактирования созданных файлов с текстом программного кода на языке ANSI C++ служит встроенный текстовый редактор, который можно открыть с помощью операции контекстного меню **ANSI C++ → Browse Header** (Язык ANSI C++ → Просмотреть заголовочный файл) или **ANSI C++ → Browse Body** (Язык ANSI C++ → Просмотреть файл реализации).

Применительно к рассматриваемой модели системы управления банкоматом программного кода на языке ANSI C++ выберем компонент с именем `Transactions`. После генерации программного кода в выбранной папке появятся 8 файлов с текстом кода на языке ANSI C++. При этом каждому классу, например, `UstroistvoChteniya`, реализованному на этом компоненте, будет соответствовать два файла — заголовочный, с расширением h (рис. 14.18), и файл реализации с расширением cpp (см. рис. 14.19).

Как видно из рассмотрения полученного заголовочного файла, в нем содержится объявление, в соответствии с правилами синтаксиса языка ANSI C++, всех атрибутов и операций класса `UstroistvoChteniya`.

UstroisrvoChtenie.h

```
#ifndef USTROISTVOCHTENIE_H_HEADER_INCLUDED_C01AC002
#define USTROISTVOCHTENIE_H_HEADER_INCLUDED_C01AC002

//##ModelId=3FC3176B032C
//##Documentation
//## Ustanavlivaetsya na bankomate
class UstroistvoChteniya
{
  public:
    //##ModelId=3FC32346033C
    int prochitatPINkod();

    //##ModelId=3FC4862A0186
    int prochitatNomerScheta();

    //##ModelId=3FC4864A00AB
    bool izvlechCard();

  private:
    //##ModelId=3FC3190A0119
    //##Documentation
    //## frozen
    int znacheniePINkoda;

    //##ModelId=3FC4860F0251
    int nomerScheta;

};
```

UstroisrvoChtenie.h | UstroisrvoChtenie.cpp

Рис. 14.18. Вид встроенного текстового редактора с загруженным в него заголовочным файлом `UstroistvoChtenie.h`

```
UstroisrvoChtenie.cpp
#include "UstroisrvoChtenie.h"

//##ModelId=3FC32346033C
int UstroistvoChteniya::prochitatPINkod()
{
}

//##ModelId=3FC4862A0186
int UstroistvoChteniya::prochitatNomerScheta()
{
}

//##ModelId=3FC4864A00AB
bool UstroistvoChteniya::izvlechCard()
{
}
UstroistvoChtenie.h   UstroistvoChtenie.cpp
```

Рис. 14.19. Вид встроенного текстового редактора с загруженным в него файлом реализации `UstroistvoChtenie.cpp`

Как видно из рис. 14.19, в файле UstroistvoChtenie.cpp содержится заготовка для реализации в соответствии с правилами синтаксиса языка ANSI C++ всех операций класса `UstroistvoChteniya`. При этом, каждая из операций имеет пустое тело реализации, которое должен написать программист самостоятельно, исходя из функциональных требований модели и синтаксиса языка программирования ANSI C++. Данную работу удобнее выполнять в среде программирования, например, MS Visual C++ или Borland C++. При использовании среды программирования, кроме компиляции, отладки и тестирования исходных модулей программы разработчик получает возможность дополнить приложение графическим интерфейсом, необходимым для взаимодействия с пользователем.

Примечание

Следует заметить, что при установленной на компьютер разработчика среде программирования, сгенерированные файлы с текстом программного кода автоматически открываются в ней после двойного щелчка на пиктограмме этих файлов. Тем не менее, лучше копировать содержимое этих файлов в предварительно созданные программные проекты в этих средах для полного контроля собственно процесса программирования.

Сгенерированные файлы с текстом программного кода содержат минимум информации. Для включения дополнительных элементов в программный

код следует изменить свойства генерации программного кода, установленные по умолчанию. Напомним, что изменение этих свойств описано ранее при выполнении этапа "Установка свойств генерации программного кода". Сгенерировать файлы с текстом программного кода при различных значениях свойств выбранного языка программирования предлагается читателям самостоятельно в качестве упражнения.

В заключение следует отметить, что эффект от использования средства IBM Rational Rose 2002 проявляется при разработке масштабных проектов. Действительно, при рассмотрении модели системы управления банкоматом может сложиться впечатление о том, что написать и отладить соответствующую программу гораздо проще непосредственно в той или иной интегрированной среде программирования.

Однако ситуация покажется не столь тривиальной, когда необходимо выполнить проект с несколькими десятками вариантов использования и сотней классов. Именно для подобных проектов явно выявляется преимущество использования средства IBM Rational Rose 2002 и нотации языка UML для документирования и реализации соответствующей модели.

Заключение

В настоящее время полностью специфицирована и документирована версия 1.5 языка UML и продолжается дальнейшая работа по его развитию. Хотя уже анонсирована следующая версия языка UML 2.0, на момент написания книги окончательная документация по этой версии еще не утверждена. Ход этой работы и ее состояние отражаются на официальном сайте консорциума OMG **www.omg.org**. Там же содержатся полные спецификации стандарта OMG-UML, предоставленные для свободного доступа.

Другим источником информации по языку UML в Интернете является сайт компании IBM Rational Software Corp. **www.rational.com**, со стороны которой осуществляется общая координация работы над очередными версиями языка UML. Эта компания по-прежнему продолжает разработку CASE-средства Rational Rose, в котором реализуются текущие дополнения языка UML.

В последнее время появилось множество программных средств, позволяющих преобразовывать форматы представления канонических диаграмм языка UML. Так, например, на сайте **www.reischmann.com** содержится информация о нескольких десятках подобных программ, реализующих преобразование графических моделей в нотации языка UML между популярными CASE-средствами.

Из отечественных ресурсов следует упомянуть сайт компании Интерфейс — **www.interface.ru**, где содержится информация по многим современным CASE-средствам, рассматриваются их характеристики и возможности, а также особенности отдельных технологий ООАП.

Перспективы дальнейшего развития UML связаны со становлением и интенсивным развитием новой парадигмы объектно-ориентированного анализа — компонентной разработки приложений (CBD, Component-Based Development). В этой связи развернута работа над дополнительной спецификацией языка UML применительно к технологиям CORBA и COM+.

Речь идет о разработке так называемых *профилей*, содержащих нотацию всех необходимых элементов для представления в языке UML компонентов соответствующих технологий. При этом интенсивно используется механизм расширения языка UML за счет добавления новых стереотипов, помеченных значений и ограничений.

Язык UML уже сейчас находит широкое применение в качестве неофициального стандарта в процессе разработки программных систем, связанных с такими областями, как моделирование бизнеса, управление требованиями, анализ и проектирование, программирование и тестирование. Применительно к этим процессам в языке UML унифицированы стандартные обозначения основных элементов соответствующих предметных областей.

Следует также отметить, что развитие языка UML на основе включения в его нотацию дополнительных элементов и стереотипов стимулирует разработку соответствующих инструментальных CASE-средств. Можно с уверенностью предположить, что эта область развития информационных технологий имеет широчайшие перспективы и стратегическое значение не только в качестве языка общения между заказчиками и разработчиками программных систем, но и для документирования проектов в целом. При этом достигается требуемый уровень стандартизации и унификации всех используемых для этой цели обозначений.

Разработав модель и специфицировав ее на языке UML, разработчик имеет все основания быть понятым и по достоинству оцененным своими коллегами. При этом невозможны ситуации, когда тот или иной разработчик применяет свою собственную графическую нотацию для представления тех или иных аспектов модели, что практически исключает ее понимание другими специалистами в случае нетривиальности исходной модели.

Не менее важный аспект применения языка UML связан с профессиональной подготовкой соответствующих специалистов. Речь идет о том, что знания различных научных дисциплин характеризуют различные аспекты реального мира. При этом принципы системного анализа позволяют рассматривать те или иные объекты в качестве систем.

Последующая разработка модели системы, направленная на решение определенных проблем, может потребовать привлечения знаний из различных дисциплин. С этой точки зрения язык UML может быть использован не только для унификации представлений этих знаний, но что не менее важно — для их интеграции, направленной на повышение адекватности многомодельных представлений сложных систем.

Уже сейчас язык UML становится тем "эсперанто", на котором могут общаться между собой руководители информационных служб, менеджеры проектов, системные аналитики, бизнес-аналитики, программисты, тести-

ровщики, математики, экономисты, физики и специалисты других профессий, представляя свои профессиональные знания в унифицированном виде. Ведь, по существу, каждый из специалистов оперирует модельными представлениями в своей области знаний. И именно этот модельный аспект может быть специфицирован и документирован средствами языка UML.

В связи с этим значение языка UML неуклонно возрастает, поскольку он все более приобретает черты языка представления знаний. При этом наличие в языке UML средств для изображения структуры и поведения модели позволяет достичь адекватного представления декларативных и процедурных знаний и, что не менее важно — установить между этими формами знаний семантическое соответствие. Все эти особенности языка UML позволяют сделать вывод о том, что в ближайшем будущем он займет лидирующее положение среди графических нотаций моделирования и документирования программных систем и бизнес-процессов.

Приложение

Язык объектных ограничений

Язык объектных ограничений (Object Constraint Language, OCL) представляет собой некоторый формальный язык для описания ограничений, которые могут быть использованы при определении различных компонентов языка UML. Хотя язык OCL является частью языка UML, в общем случае он может иметь гораздо более широкую область приложений, чем конструкции языка UML. Средства языка OCL позволяют специфицировать не только объектные ограничения, но и другие выражения логико-лингвистического характера, такие как предусловия, постусловия и сторожевые условия. Все это делает его чрезвычайно мощным инструментом системного моделирования.

В процессе ООАП одних графических средств, таких как диаграмма классов или диаграмма состояний, недостаточно для разработки точных и непротиворечивых моделей сложных систем. Существует необходимость задания дополнительных ограничений, которым должны удовлетворять различные компоненты или объекты модели. Традиционно подобные ограничения описывались с помощью естественного языка. Однако, как показала практика системного моделирования и ООАП, такие сформулированные на естественном языке ограничения страдают неоднозначностью и нечеткостью.

Для преодоления этих недостатков и придания рассуждениям строгого характера были разработаны различные *формальные* языки. Однако традиционные формальные языки требуют для своего конструктивного использования знания основ математической логики и теории формального вывода. С одной стороны, это делает формальные языки естественным средством для построения математических моделей. С другой стороны, существенно ограничивает круг потенциальных пользователей, поскольку разработка формальных моделей требует специальной квалификации.

Другая особенность традиционных формальных языков заключается в присущей им "бедной" семантике. Речь идет о том, что базовые элементы формальных языков имеют слишком абстрактное содержание, что затрудняет их непосредственную интерпретацию в понятиях моделей конкретных технических

систем и бизнес-процессов. Поскольку процесс ООАП направлен на построение конструктивных моделей сложных систем, которые должны быть реализованы в форме программных систем и аппаратных комплексов, это является серьезным недостатком. Речь идет о необходимости применения такого формального языка, базовые элементы которого адекватно отражают семантику основных конструкций ООАП.

Именно для этих целей и был разработан язык OCL. По своей сути он является формальным языком, более простым для изучения, чем традиционные формальные языки. В то же время язык OCL специально ориентирован на описание бизнес-процессов и бизнес-логики, поскольку был разработан в одном из отделений корпорации IBM Джосом Уормером (Jos Warmer). Язык OCL представляет собой дальнейшее развитие метода Syntropy Стива Кука (Steve Cook) и Джона Дэниэлса (John Daniels). Язык OCL является языком формальных выражений, которые используются для записи отдельных ограничений. Этот язык ничего не изменяет в графических моделях, а только дополняет их. Это означает, что выражения языка OCL никогда не могут изменить состояние системы, хотя эти выражения и могут быть использованы для спецификации самого процесса изменения этого состояния. Выражения языка OCL также не могут изменять отдельные значения атрибутов и операций для объектов и их связей. Всякий раз, когда оценивается некоторое такое выражение, на выходе получается лишь некоторое значение.

Язык OCL не является языком программирования в обычном смысле, поскольку не позволяет записать логику выполнения программы или последовательность управляющих действий. Средства этого языка не предназначены для описания процессов вычисления выражений, а только лишь фиксируют необходимость выполнения тех или иных условий применительно к отдельным компонентам моделей ООАП. Главное назначение языка OCL — гарантировать справедливость ограничений для отдельных объектов модели. В этом смысле OCL можно считать языком моделирования, который вовсе не обязан допускать свою строгую реализацию в инструментальных средствах.

Примечание

Корпорацией IBM разработана и свободно распространяется программа для синтаксического разбора OCL-выражений (OCL Parser). Исходный код этой программы написан на языке Java и доступен в Интернете по адресу: **http://www-4.ibm.com/software/ad/standards/ocl-parser**.

Язык OCL может быть использован для решения следующих задач:

- описание инвариантов классов и типов в модели классов;
- описание пред- и постусловий в операциях и методах;
- описание сторожевых условий элементов модели;
- навигация по структуре модели;
- спецификация ограничений на операции.

Язык OCL также применяется для спецификации базовой нотации в метамодели языка UML, в частности, для записи правил правильного построения выражений, определяющих семантику элементов основных пакетов языка UML.

Язык OCL является *типизированным* языком, поскольку каждое выражение языка OCL имеет некоторый тип. В правильно построенном выражении языка OCL все типы должны быть совместимы между собой. Например, нельзя в выражении сравнивать целое число со строкой. Типом может быть любой вид классификатора (classifier) языка UML.

Предполагается, что оценка выражений OCL выполняется мгновенным образом. Это означает, что состояние элементов модели не может изменяться в течение оценки того или иного выражения.

Описание языка OCL отличается от описаний традиционных формальных языков своим менее формальным характером. Базовым элементом языка OCL является выражение, которое строится по определенным правилам. При этом язык допускает расширение своих конструкций за счет включения в свой состав дополнительных типов.

Выражения языка OCL

Выражение как элемент языка OCL является составной частью метамодели и представляет собой утверждение относительно множества (возможно, пустого) экземпляров некоторого объекта языка UML. Выражение OCL является контекстно-зависимым в том смысле, что каждое выражение рассматривается только в том контексте или окружении, в котором определен соответствующий объект UML. С другой стороны, выражение не может модифицировать этот контекст или окружение объекта.

В модели выражение используется для записи некоторых условий, которым должны удовлетворять все экземпляры соответствующего классификатора. В этом случае говорят, что выражение служит для представления некоторых инвариантных свойств соответствующих элементов модели.

Выражение состоит из тела и имени языка интерпретации, в рамках которого оценивается тело данного выражения. *Тело* выражения представляет собой строку текста без кавычек на некотором языке, возможно, некотором подмножестве естественного языка. Для записи выражения OCL используется моноширинный шрифт. Например, строка текста, взятая без кавычек: `'Эта строка текста есть выражение языка OCL'` по определению является выражением OCL.

При записи выражений могут использоваться некоторые ключевые слова. Часть из них записывается полужирным шрифтом на языке оригинала. Например, ключевое слово `context` представляет контекст выражения. Ключевые

слова `inv`, `pre` и `post` обозначают стереотипы, соответственно "инвариант", "предусловие" и "постусловие" некоторого ограничения. Фактическое выражение OCL записывается после двоеточия. Например:

```
context TypeName inv:
      'Это выражение OCL со стереотипом «инвариант» в контексте
      TypeName′ =′ другая строка'
```

Рекомендуется записывать выражения OCL, используя только символы кодировки ASCII.

Каждое выражение OCL должно быть записано в контексте некоторого экземпляра специфического типа. Для обращения к контексту известного экземпляра может быть использовано ключевое слово `self`. Как правило, из записи выражения непосредственно следует, о контексте какого экземпляра идет речь. Например, если контекст выражения — компания, то `self` относится к отдельному экземпляру компании.

Контекст выражения OCL внутри UML-модели может быть определен через так называемое *объявление контекста* в начале выражения OCL. Объявление контекста ограничения на диаграмме может быть представлено соответствующим стереотипом и пунктирной линией, связывающей это ограничение с контекстом.

Выражение OCL может быть частью записи некоторого инвариантного свойства или инварианта, записанное в форме ограничения со стереотипом "инвариант". Выражение OCL, которое является инвариантом типа, должно быть истинно для всех экземпляров классификатора этого типа. Следует заметить, что все выражения OCL, которые выражают инварианты, имеют Булев тип, т. е. при своей оценке могут принимать значения "истина" или "ложь".

Предположим, что в контексте некоторой компании (типа компании) следует определить инвариант, который устанавливает обязательное условие — число служащих любой отдельно взятой компании этого типа всегда должно превышать 50. Это можно записать в форме следующего выражения OCL:

```
self.numberOfEmployees > 50
```

Однако при записи инварианта в некотором контексте должно быть указано соответствующее ключевое слово `inv`, которое объявляет, что следующее за ним ограничение будет "инвариантным" ограничением. Окончательная запись инвариантного ограничения будет иметь следующий вид:

```
context Компания inv:
      self.numberOfEmployees > 50
```

В большинстве случаев ключевое слово `self` может быть пропущено, потому что контекст очевиден, как в этом примере. Как вариант записи можно использовать явное определение объекта, в контексте которого записывается выражение:

```
context c : Компания inv:
      c.numberOfEmployees > 50
```

Этот инвариант эквивалентен предыдущему. Дополнительно, но необязательно, после ключевого слова `inv` и перед двоеточием может быть написано имя ограничения, что позволяет ссылаться на это ограничение из других элементов модели. Так, если определить имя этого ограничения как "Персонал", то оно может быть записано в следующем виде:

```
context c : Компания inv Персонал:
      c.numberOfEmployees > 50
```

Основные типы значений и операций в языке OCL

В языке OCL ряд основных или базисных типов предопределен и всегда доступен разработчику модели. Эти предопределенные типы значений не зависят от конкретной объектной модели и не могут быть переопределены, поскольку являются частью определения языка OCL.

Основные типы значений языка OCL перечислены далее.

- *Булев тип*. Принимает значения "истина" или "ложь".
- *Целое число*. Может принимать любое значение из множества целых чисел, например: -5, 2, 34, 26524.
- *Действительное число*. Может принимать любое значение из множества действительных чисел, например: 1,5; 3,14.
- *Строка*. Содержит любое количество символов текста, например: `'быть или не быть ... '`.

В языке OCL также определены операции над этими основными типами, некоторые из них представлены ниже.

- Для Булева типа определены операции: `and`, `or`, `xor`, `not`, `implies`, `if-then-else-endif`.
- Для целых чисел определены арифметические операции: `+`, `-`, `*`, `/`, `div` и операции `abs`, `mod`, `max`, `min`.
- Для действительных чисел определены арифметические операции: `+`, `-`, `*`, `/` и операции `abs`, `floor`, `round`, `max`, `min`.
- Для типа *строка* определены операции конкатенации `concat`, определения размера `size`, преобразования регистра `toUpper` и `toLower`.

В выражениях языка OCL также используются операции сравнения и круглые скобки для указания последовательности выполнения этих операций. Последовательность выполнения операций определяется их приоритетом. По умолчанию в языке OCL установлен следующий приоритет операций (в порядке убывания приоритета):

- @pre
- точка "`.`" или операция *стрелка* "`->`"

- унарное логическое отрицание `not` и унарный минус "`-`"
- умножение "`*`" и деление "`/`"
- сложение "`+`" и бинарный минус "`-`"
- логический условный оператор "`if-then-else-endif`"
- операции сравнения "`<`", "`<=`", "`>`", "`>=`"
- операции равенства "`=`" и неравенства "`<>`"
- логические бинарные операции `and`, `or` и `xor`
- логическая бинарная операция `implies`

Для записи комментария применяется символ двойного минуса "`--`". Например, следующая запись является комментарием языка OCL:

```
-- Это комментарий языка OCL
```

Операции над отдельными типами значений

Большинство допустимых операций в языке OCL аналогичны известным математическим операциям и имеют соответствующие названия. Поэтому ниже представлена лишь краткая характеристика этих операций и приведены простейшие примеры их применения.

В языке OCL в качестве основной используется *инфиксная* запись операций. Это означает, что второй аргумент бинарной операции рассматривается как ее параметр, а сама операция применяется к первому аргументу. При этом применение операции к этому аргументу обозначается точкой после этого аргумента. Таким образом, выражение `a+b` эквивалентно следующему выражению языка OCL: `a.+(b)`, где второй аргумент `b` выступает в роли параметра операции сложения. Тем не менее, допускается использование и обычной записи, если это не вызывает недоразумений.

Операции с действительными числами

Действительные числа являются наиболее общим типом чисел, которые используются в языке OCL. Формальная запись отдельной бинарной операции имеет следующий общий формат:

```
r1 'символ операции' r2 :'тип возвращаемого значения'
```

или эквивалентная запись:

```
r1.'символ операции'(r2) :'тип возвращаемого значения'.
```

Здесь `r1`, `r2` обозначают произвольные действительные числа, `'символ операции'` — условное обозначение операции, а `'тип возвращаемого значения'` должен соответствовать одному из рассмотренных ранее допустимых типов языка OCL.

Прежде всего, для действительных чисел определены некоторые операции сравнения.

- `r1 = r2 : Boolean`. Возвращает значение "истина", если значение `r1` в точности равно значению `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.14 = 3.14` принимает значение "истина", а выражение `3.14 = 3.1` принимает значение "ложь".
- `r1 <> r2 : Boolean`. Возвращает значение "истина", если значение `r1` не равно значению `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.14 <> 3.1` принимает значение "истина", а выражение `3.14 <> 3.14` принимает значение "ложь".
- `r1 < r2 : Boolean`. Возвращает значение "истина", если значение `r1` строго меньше значения `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.1 < 3.14` принимает значение "истина", а выражение `3.14 < 3.1` принимает значение "ложь".
- `r1 > r2 : Boolean`. Возвращает значение "истина", если значение `r1` строго больше значения `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.14 > 3.1` принимает значение "истина", а выражение `3.1 > 3.14` принимает значение "ложь".
- `r1 <= r2 : Boolean`. Возвращает значение "истина", если значение `r1` меньше или равно значению `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.1 <= 3.14` принимает значение "истина", а выражение `3.14 < = 3.1` принимает значение "ложь".
- `r1 >= r2 : Boolean`. Возвращает значение "истина", если значение `r1` больше или равно значению `r2`, в противном случае возвращает значение "ложь". Например, выражение `3.14 >= 3.14` принимает значение "истина", а выражение `3.1 >= 3.14` принимает значение "ложь".

Далее для действительных чисел определены некоторые арифметические операции.

- `r1 + r2 : Real`. Возвращает результат сложения двух действительных чисел. Например, выражение `3.14 + 3.14` принимает значение "6.28".
- `r1 - r2 : Real`. Возвращает результат вычитания двух действительных чисел. Например, выражение `3.14 - 3.14` принимает значение "0".
- `r1 * r2 : Real`. Возвращает результат умножения двух действительных чисел. Например, выражение `3.14 * 3.14` принимает значение "9.8596".
- `r1 / r2 : Real`. Возвращает результат деления двух действительных чисел. Например, выражение `3.14 / 3.14` принимает значение "1".

И, наконец, имеются некоторые дополнительные операции, которые могут применяться к действительным числам (здесь `r` — любое действительное число).

- `r.abs : Real`. Возвращает в качестве результата абсолютное значение действительного числа `r`. Другими словами, возвращает само значение `r`,

если `r>=0`, и возвращает значение `-r`, если `r<0`. Например, выражение `3.14 .abs` принимает значение "3.14", а выражение `(3.14-5.14).abs` принимает значение "2".

- `r.floor : Integer`. Возвращает в качестве результата наибольшее целое число, которое меньше или равно значению действительного числа `r`. Другими словами, возвращаемое значение меньше или равно `r` и одновременно строго больше `r-1`. Например, выражение `3.14 .floor` принимает значение "3", а выражение `(3.14-5.1).floor` принимает значение "-2".
- `r.round : Integer`. Возвращает в качестве результата целое число, которое является ближайшим к значению действительного числа `r`. Если такими целыми являются сразу два числа, то возвращается наибольшее из них. Другими словами, после этой операции должно выполняться следующее постусловие: `post: ((r-result).abs<0.5) or ((r-result).abs=0.5 and (result>r))`. Здесь `result` служит для обозначения возвращаемого данной операцией значения. Например, выражение `3.14.round` принимает значение "3", а выражение `(3.14-5.64).round` принимает значение "-2".
- `r1.max(r2) : Real`. Возвращает в качестве результата наибольшее из двух действительных чисел `r1` или `r2`. Другими словами, после этой операции должно выполняться следующее постусловие: `post:` если `r1>=r2`, то `result=r1`, иначе `result=r2`. Здесь `result` также служит для обозначения возвращаемого данной операцией значения. Для записи подобных условных выражений в языке OCL используется специальная конструкция `if-then-else-endif`, которая будет рассмотрена ниже. Например, выражение `(3.14).max(3.1)` принимает значение "3.14".
- `r1.min(r2) : Real`. Возвращает в качестве результата наименьшее из двух действительных чисел `r1` или `r2`. Другими словами, после этой операции должно выполняться следующее постусловие: `post:` если `r1<=r2`, то `result=r1`, иначе `result=r2`. Здесь `result` также служит для обозначения возвращаемого данной операцией значения. Например, выражение `(3.14).min(3.1)` принимает значение "3.1".

Поскольку целые числа в языке OCL являются подтипом действительных чисел, то в рассмотренные выше операции в качестве действительных параметров могут входить отдельные целочисленные значения, что не противоречит общему определению этих операций.

Операции с целыми числами

Хотя целые числа являются подтипом действительных чисел, для них вводятся в рассмотрение самостоятельные операции, для формальной записи которых используется аналогичный общий формат (для бинарных операций):

```
i1 'символ операции' i2 :'тип возвращаемого значения'
```

или эквивалентная запись:

`i1.‘символ операции‘(i2) :’тип возвращаемого значения’.`

Здесь `i1`, `i2` обозначают произвольные целые числа, `‘символ операции‘` — условное обозначение операции, а `’тип возвращаемого значения’` должен соответствовать одному из рассмотренных ранее допустимых типов языка OCL.

Прежде всего, для действительных чисел определена операция равенства:

- `i1 = i2 : Boolean`. Возвращает значение "истина", если значение `i1` в точности равно значению `i2`, в противном случае возвращает значение "ложь". Например, выражение `3 = 3` принимает значение "истина", а выражение `3 = 2` принимает значение "ложь".

Далее для целых чисел определены следующие арифметические операции:

- `i1 + i2 : Integer`. Возвращает результат сложения двух целых чисел. Например, выражение `3 + 2` принимает значение "5";
- `i1 - i2 : Integer`. Возвращает результат вычитания двух целых чисел. Например, выражение `3 - 4` принимает значение "-1";
- `i1 * i2 : Integer`. Возвращает результат умножения двух целых чисел. Например, выражение `3 * 4` принимает значение "12"%;
- `i1 / i2 : Real`. Возвращает результат деления двух целых чисел в виде действительного числа. Например, выражение `3 /2` принимает значение "1.5".

И, наконец, имеются некоторые дополнительные операции, которые могут применяться к целым числам (здесь `i` — любое целое число).

- `i.abs : Integer`. Возвращает в качестве результата абсолютное значение целого числа `i`. Другими словами, эта операция возвращает само значение `i`, если `i>=0`, и возвращает значение `-i`, если `i<0`. Например, выражение `3.abs` принимает значение "3", а выражение `(3-5).abs` принимает значение "2".
- `i1.div(i2) : Integer`. Возвращает в качестве результата целое число, равное частному от деления `i1` на `i2` с отброшенной дробной частью. Другими словами, перед выполнением этой операции должно быть справедливым предусловие: `pre i2<>0` (делитель не должен быть равен 0), а после выполнения этой операции — постусловие: `post`: если `i1/i2>=0`, то `result=(i1/i2).floor`, иначе `result=-((-i1/i2).floor)`. Здесь `result`, как и ранее, служит для обозначения возвращаемого данной операцией значения. Например, выражение `3.div( 2)` принимает значение "1".
- `i1.mod(i2) : Integer`. Возвращает в качестве результата целое число, равное остатку от деления `i1` на `i2`. Другими словами, после выполне-

ния этой операции должно выполняться следующее постусловие: `post: result=i1-(i1.div(i2))*i2`. Например, выражение `5.mod( 2)` принимает значение "1".

- `i1.max(i2) : Integer`. Возвращает в качестве результата наибольшее из двух целых чисел `i1` или `i2`. Другими словами, после выполнения этой операции должно выполняться следующее постусловие: `post:` если `i1>=i2`, то `result=i1`, иначе `result=i2`. Например, выражение `3.max(1)` принимает значение "3".
- `i1.min(i2) : Integer`. Возвращает в качестве результата наименьшее из двух целых чисел i1 или i2. Другими словами, после выполнения этой операции должно выполняться следующее постусловие: `post:` если `i1<=i2`, то `result=i1`, иначе `result=i2`. Например, выражение `3.min(1)` принимает значение "1".

Операции со строками

Тип строки в языке OCL соответствует ASCII строкам, т. е. допускается в качестве отдельных символов строк использовать только символы кодировки ASCII. В этой связи резонно предположить, что имеется в виду расширенная кодировка ASCII, которая допускает применение символов кириллицы, хотя этот факт никак не отражается в исходной документации.

Для формальной записи бинарных операций со строками используется общий формат:

```
s1 'символ операции' s2 :'тип возвращаемого значения'
```

или эквивалентная запись:

```
s1.'символ операции'(s2) :'тип возвращаемого значения'.
```

Здесь `s1`, `s2` обозначают произвольные строки, `'символ операции'` — условное обозначение операции, а `'тип возвращаемого значения'` должен соответствовать одному из рассмотренных ранее допустимых типов языка OCL.

Прежде всего, для строк определена операция равенства.

- `s1 = s2 : Boolean`. Возвращает значение "истина", если строка `s1` содержит в точности те же символы (включая их порядок и регистр), что и строка `s2`, в противном случае возвращает значение "ложь". Например, выражение `'Компания' = 'Компания'` принимает значение "истина", а выражение `'Компания' ='компания'` принимает значение "ложь".

Далее для строк определены следующие унарные операции:

- `s.size : Integer`. Возвращает в качестве результата количество символов в строке `s`. При этом пробелы также должны учитываться. Например, выражение `'Компания'.size` принимает значение "8".

- `s.toUpper : String`. Возвращает в качестве результата строку, в которой все символы (буквы) нижнего регистра исходной строки `s` заменены аналогичными буквами верхнего регистра. При этом должно выполняться следующее постусловие: `post: result.size = s.size`. Здесь `result`, как и ранее, служит для обозначения возвращаемого данной операцией значения. Например, выражение `'Компания'.toUpper` принимает значение "'КОМПАНИЯ'".
- `s.toLower : String`. Возвращает в качестве результата строку, в которой все символы (буквы) верхнего регистра исходной строки `s` заменены аналогичными буквами нижнего регистра. При этом должно выполняться следующее постусловие: `post: result.size = s.size`. Например, выражение `'Компания'.toLower` принимает значение "'компания'".

И, наконец, для строк определены бинарная операция *конкатенации* и тернарная операция выделения подстроки заданной длины (здесь `i1`, `i2` — произвольные целые числа).

- `s1.concat(s2) : String`. Возвращает в качестве результата строку, в которой вначале следуют все символы первой строки `s1`, а затем без пробела — все символы второй строки `s2`. При этом должны выполняться следующие постусловия: `post: result.size = s1.size+s2.size; post: result.substring(1, s1.size)= s1; post: result.substring(s1.size+1, result.size)= s2`. Здесь используется операция выделения подстроки заданной длины `substring`, которая определяется ниже. Например, выражение `'Персональный'.concat(' Компьютер')` принимает значение "'Персональный Компьютер'".
- `s.substring(i1, i2) : String`. Возвращает в качестве результата строку, которая является подстрокой исходной строки `s`, начиная символом с номером `i1` и заканчивая символом с номером `i2`. Например, выражение `'Персональный'.substring(1, 7)` принимает значение "'Персона'". Параметрами этой операции являются два целых числа, указывающие порядковый номер соответствующих символов.

Операции с Булевыми выражениями

Напомним, что Булевы выражения могут принимать только одно их двух возможных значений: "истина" или "ложь". Для формальной записи бинарных операций с Булевыми выражениями используется обычный не инфиксный формат, принятый в математической логике (`b`, `b1`, `b2` — произвольные Булевы выражения).

Прежде всего, для Булевых выражений определена бинарная операция равенства:

- `b1=b2: Boolean`. Возвращает в качестве результата значение "истина", если значения `b1` и `b1` одновременно принимают одно из значений —

"истина" или "ложь", и значение "ложь" — в противном случае. Например, выражение `(5<3) = (2=3)` принимает значение "истина", а выражение `(5<3) = (2<>3)` принимает значение "ложь".

Далее, для Булевых выражений определена унарная операция *логического отрицания*:

- `not b: Boolean`. Возвращает в качестве результата значение "истина", если значение `b` — "ложь", и значение "ложь", если значение `b` — "истина". Другими словами, после выполнения этой операции должно выполняться следующее постусловие: `post:` если `b= "истина"`, то `result="ложь"`, иначе `result="истина"`. Например, выражение `not (5<3)` принимает значение "истина".

Для Булевых выражений определены следующие основные логические (бинарные) операции.

- `b1 or b2: Boolean`. Логическое "ИЛИ". Возвращает в качестве результата значение "истина", если значение "истина" имеет или `b1`, или `b2`, или они одновременно, и значение "ложь" — в противном случае. Например, выражение `(5<3) or (2=3)` принимает значение "ложь", а выражение `(5<3) or (2<>3)` принимает значение "истина".
- `b1 xor b2: Boolean`. Логическое "исключающее ИЛИ". Возвращает в качестве результата значение "истина", если значение "истина" имеет или `b1`, или `b2` (но не одновременно), и значение "ложь" — в противном случае. Для этой операции должно выполняться следующее постусловие: `post: (b1 or b2) and not(b1 = b2)`. Здесь используется операция логического "И" (`and`), которая определяется ниже. Например, выражение `(5<3) xor (2=3)` принимает значение "ложь", а выражение `(5<3) xor (2<3)` принимает значение "истина".
- `b1 and b2: Boolean`. Логическое "И". Возвращает в качестве результата значение "истина", если значение "истина" имеют `b1` и `b1` одновременно, и значение "ложь" — в противном случае. Например, выражение `(5<3) and (2=3)` принимает значение "ложь", а выражение `(5>3) and (2<>3)` принимает значение "истина".

И, наконец, для Булевых выражений определены бинарная операция логической *импликации* и упоминавшаяся ранее специальная логическая конструкция `if-then-else-endif` (`e1` и `e2` — произвольные выражения языка OCL).

- `b1 implies b2: Boolean`. Логическая импликация. Возвращает в качестве результата значение "истина", если `b1` имеет значение "ложь" или если `b1` и `b2` имеют значение "истина" одновременно, и значение "ложь" — в противном случае. Например, выражение `(5<3) implies (2=3)` принимает значение "истина", а выражение `(5>3) implies (2=3)` принимает значение "ложь".

- `if b then e1 else e2 endif : тип e1 или e2`. Логическая условная конструкция. Возвращает в качестве результата выражение `e1`, если Булево выражение `b` имеет значение "истина", и возвращает в качестве результата выражение `e2`, если Булево выражение `b` имеет значение "ложь". Например, рассмотренное ранее постусловие для логического отрицания `post:` если `b="истина"`, то `result="ложь"`, иначе `result="истина"` может быть записано с использованием данной конструкции в следующем виде : `post: if b= "истина" then result="ложь" else result="истина" endif.`

Операция @pre для указания предшествующих элементов

Особенность записи постусловий для операций заключается в том, что они могут ссылаться на значения элементов языка OCL, которые имели место до завершения соответствующей операции или метода. В этом случае возникает необходимость явно указать то значение элемента, которое он имел до начала выполнения операции. Для этой цели в языке OCL используется специальное постфиксное ключевое слово `@pre`. Постфиксный характер этого слова состоит в том, что оно всегда записывается после имени элемента без пробела.

Например, если необходимо записать постусловие для операции определения возраста сотрудника компании в текущем году, то это можно сделать следующим образом:

```
context Сотрудник::Возраст()
post: if t_Date >= b_Date then
      возраст = возраст@pre+1
      else возраст = возраст@pre endif.
```

Здесь `t_Date` обозначает текущую дату (день, месяц), `b_Date` — дату рождения сотрудника (день, месяц). Предполагается, что обе эти даты преобразованы к типу целое число.

Допустимые выражения в языке OCL

Поскольку OCL — язык с контролем типа выражений, все его выражения должны быть согласованными с основными или производными типами значений. Например, нельзя сравнивать целое число с Булевым типом или строкой. Выражение OCL, в котором все типы согласованы между собой, называется *допустимым* выражением языка OCL. Выражение OCL, в котором некоторые типы не согласованы, — недопустимое выражение. Последний случай является ошибкой согласованности типов.

Существует два правила согласованности типов выражений.

1. Каждый тип соответствует каждому из своих супертипов.
2. Согласованность типов транзитивна: если тип 1 соответствует типу 2 и тип 2 соответствует типу 3, то тип 1 соответствует типу 3.

Примеры допустимых выражений языка OCL: "`1 + 2 * 34`", "`12 + 13.5`", "`13 < 2`".

Примеры недопустимых выражений языка OCL: "`1 + 'мотоцикл'`", "`23 * ложь`".

Неопределенные значения выражений

Всякий раз, когда выражение OCL оценивается, существует возможность, что один или несколько элементов в выражении могут быть не определены. Если дело обстоит так, то полное выражение будет иметь значение "неопределенный".

Однако из этого правила имеются два исключения для определенных логических операций.

1. Выражение принимает значение "истина", если хотя бы один Булев тип в операции `OR` принимает значение "истина".
2. Выражение принимает значение "ложь", если хотя бы один Булев тип в операции `AND` принимает значение "ложь".

Вышеупомянутые два правила применимы независимо от порядка записи параметров и являются или нет известными значения других подвыражений.

Совокупности допустимых значений в языке OCL

Язык OCL содержит развитые средства для представления различных совокупностей допустимых значений типов. При этом может учитываться порядок следования значений или повторение отдельных значений в совокупности. В языке OCL различаются три типа совокупности значений.

1. *Множество* (set) — представляет собой неупорядоченный список отдельных значений основного типа без повторяющихся элементов. Пример множества, состоящего из положительных целых чисел: `{1, 4, 6, 8, 2, 5, 10}`.
2. *Последовательность* (sequence) — представляет собой упорядоченный список отдельных значений основного типа с возможно повторяющимися элементами. Пример последовательности, состоящей из положительных целых чисел: `{1, 2, 4, 5, 6, 8, 10}`. Совокупность чисел `{0, 1, 3, 3, 7, 8, 8}` также является последовательностью.

3. *Комплект* (Bag) — представляет собой неупорядоченный список отдельных значений основного типа, элементы в котором могут повторяться. Пример комплекта, состоящего из положительных целых чисел: `{1, 2, 1, 4, 5, 8, 6, 8, 1, 10}`.

Для определения совместимости этих типов необходимо придерживаться следующего правила — типы "множество", "последовательность" и "комплект" являются подтипами типа "совокупность".

Операции над совокупностями значений

Над совокупностью значений могут быть выполнены некоторые операции, такие как операция *выбора* (select) или *исключения* (reject), а также операции `collect`, `forAll`, `exists`.

Операция выбора `select`

Операция выбора используется для спецификации части совокупности, элементы которой удовлетворяют некоторому логическому условию. Общий формат записи этой операции имеет следующий вид:

Совокупность->select (*Булево выражение*)

В результате применения этой операции возвращаются все элементы исходной совокупности, для которых Булево выражение имеет значение "истина". Например, следующее выражение языка OCL специфицирует обязательное требование, которое утверждает, что среди сотрудников компании должны быть служащие, возраст которых превышает 50 лет:

```
context Компания inv:
      self.сотрудник -> select(возраст > 50) -> не пусто
```

В этом случае операция выбора проверяет условие выбора последовательно для всех сотрудников компании. Если это условие выполняется, то сотрудник попадает в формируемую операцией выбора совокупность, в противном случае — нет. При этом ключевое слово "не пусто" (notEmpty) означает, что возвращаемая операцией совокупность должна содержать хотя бы один элемент. В противном случае она считается пустой совокупностью, по аналогии с пустым множеством.

Операция исключения `reject`

Операция исключения возвращает совокупность значений, для которых Булево выражение принимает значение "ложь". Формат записи этой операции такой же, как и для операции выбора. Например, с помощью этой операции

можно специфицировать условие, согласно которому все сотрудники компании должны быть семейными людьми:

```
context Компания inv:
      self.сотрудник -> reject(имеет семью) -> пусто
```

При этом ключевое слово "пусто" (isEmpty) означает, что возвращаемая операцией совокупность не должна содержать ни одного элемента.

Операция формирования совокупности `collect`

Рассмотренные выше операции выбора и исключения всегда возвращают в качестве результата часть исходной совокупности значений (возможно — пустую). Если необходимо специфицировать совокупность значений, которая получается из некоторой другой совокупности на основе проверки условия для отдельного свойства, то используется операция `collect`. Эта операция позволяет сформировать совокупность значений, которые отличаются от значений исходной совокупности, но контекстно связаны с ней.

Формат применения этой операции следующий:

```
Исходная_совокупность->collect( выражение).
```

Например, если необходимо сформировать совокупность дат принятия на работу сотрудников компании, то это можно сделать с помощью следующего выражения языка OCL:

```
self.сотрудник->collect( Дата_принятия_на_работу).
```

В результате применения этой операции будет сформирована совокупность, значениями которой будут являться даты приема на работу сотрудников рассматриваемой компании. Следует заметить, что тип значений возвращаемой этой операцией совокупности может отличаться от типа значений исходной совокупности.

Операция "для всех" `forAll`

Некоторые типы ограничений могут относиться сразу ко всем элементам заданной совокупности. Для этого случая в языке OCL существует специальная операция `forAll`. Эта операция позволяет специфицировать Булево выражение, которое должно быть справедливым для всех элементов совокупности. Общий формат записи операции `forAll` имеет следующий вид:

```
Совокупность->forAll(Булево выражение): Boolean.
```

Эта операция возвращает значение "истина", если `Булево выражение` справедливо для всех без исключения элементов исходной совокупности, и возвращает значение "ложь" в противном случае. Операция `forAll` может быть использована для записи инвариантов. Например, если в контексте

некоторого банка счет клиента не может быть меньше 100$, то это условие (инвариант) можно записать в следующем виде:

```
context Б : Банк inv:
self.клиенты->forAll(счет >= 100$).
```

Очевидно, что если хотя бы для одного клиента это условие не будет выполнено, то данное инвариантное условие будет нарушено, что недопустимо в языке OCL. Строго говоря, операция `forAll` выполняет роль квантора математической логики, однако достоинством языка OCL является возможность ее записи именно в форме операции.

Другим примером может служить запись распространенного инварианта, который требует единственности имен, в частности — имен клиентов банка:

```
context Б : Банк inv:
self.клиенты->forAll(к1, к2 : клиент | к1<>к2 implies
к1.имя<>к2.имя).
```

Операция "существует" `exists`

Как следует из названия этой операции, она применяется для спецификации ситуации, при которой исходная совокупность содержит хотя бы один элемент, удовлетворяющий некоторому Булеву выражению. Общий формат записи операции `exists` имеет следующий вид:

```
совокупность->exists (Булево выражение): Boolean.
```

Эта операция возвращает значение "истина", если `Булево выражение` справедливо хотя бы для одного (а может быть и нескольких) элемента исходной совокупности, и возвращает значение "ложь" в противном случае. Операция `exists` также может быть использована для записи инвариантов.

Примером использования этой операции может служить ситуация, при которой определяется контрольный пакет акций некоторой компании (Акционерного общества). Если контрольный пакет характеризуется наличием у отдельного акционера 50% и более от общего числа акций, то это условие (инвариант) может быть записано в следующем виде:

```
context к : Компания inv Контрольный пакет акций:
self.акционеры->exists(a : акционер |  a.количество_акций >= 50%  от
общего числа акций).
Или более строго с использованием операций с целыми числами:
context к : Компания inv Контрольный пакет акций:
self.акционеры->exists(a : акционер |
a.количество_акций >= к.количество_акций.div(2)).
```

В этом примере инвариант имеет свое собственное имя, на которое можно ссылаться из других компонентов модели. Следует отметить, что операция

exists также выполняет роль квантора математической логики, но возможность ее записи в форме операции обладает целым рядом удобств.

Другие операции над совокупностью значений

Поскольку совокупность в языке OCL обобщает типы множества, последовательности и комплекта, применительно к этому супертипу определены три операции.

- совокупность->count(объект_OCL) : Integer. Эта операция подсчитывает количество вхождений заданного элемента в исходную совокупность. Операция count возвращает целое число, равное количеству вхождений объекта_OCL в эту совокупность. Например, если исходная совокупность равна {2, 4, 5, 4, 6, 7, 4, 9, 2}, то выражение совокупность->count(4) равно значению "3". С другой стороны, выражение совокупность->count(3) равно значению "0", поскольку число 3 не входит в эту совокупность.
- совокупность->includes(объект_OCL) : Boolean. Эта операция проверяет наличие в исходной совокупности конкретного объекта языка OCL. Операция includes возвращает значение "истина", если объект_OCL является элементом исходной совокупности, и значение "ложь" в противном случае. Для этой операции должно выполняться следующее постусловие: post: result = (совокупность->count(объект_OCL)>0).
- совокупность->excludes(объект_OCL) : Boolean. Эта операция проверяет отсутствие в исходной совокупности конкретного объекта языка OCL. Операция excludes возвращает значение "истина", если объект_OCL не является элементом исходной совокупности, и значение "ложь" в противном случае. Для этой операции должно выполняться следующее постусловие: post: result = (совокупность->count(объект_OCL)=0).

Некоторые операции с множествами, последовательностями и комплектами

Для множеств в языке OCL определены основные теоретико-множественные операции: объединение, пересечение, разность множеств *(см. главу 2)*. При определении этих операций set1, set2, set3 обозначают произвольные множества языка OCL.

- set1->union(set2) : set3. Операция *объединения* множеств. В качестве результата возвращает множество set3, которое представляет собой теоретико-множественное объединение исходных множеств set1 и set2. Множество set3 состоит из элементов, которые принадлежат или множеству set1, или множеству set2, или им обоим.

- `set1->intersection(set2) : set3`. Операция *пересечения* множеств. В качестве результата возвращает множество `set3`, которое представляет собой теоретико-множественное пересечение исходных множеств `set1` и `set2`. Множество `set3` состоит из элементов, которые одновременно принадлежат множеству `set1` и множеству `set2`.
- `set1 - set2 : set3`. Операция *разности* множеств. В качестве результата возвращает множество `set3`, которое представляет собой теоретико-множественную разность исходных множеств `set1` и `set2`. Множество `set3` состоит из элементов, которые принадлежат или множеству `set1` и не принадлежат множеству `set2`.

Аналогичные операции определяются применительно к последовательностям и комплектам, однако их семантика является более сложной. В частности, операции пересечения и объединения последовательностей имеют несколько постусловий, каждое из которых отражает отдельный аспект получаемых результатов.

Операции преобразования типов

Поскольку множество, последовательность и комплект являются подтипами типа совокупность, некоторые из них можно преобразовывать друг в друга. Речь идет о том, что каждая конкретная совокупность может быть записана в форме множества, хотя при этом возможна некоторая потеря информации. В форме множества может быть записана совокупность и комплект. Ниже приводятся операции, которые позволяют преобразовывать различные подтипы совокупности.

- `комплект->asSet : set`. Операция `asSet` позволяет преобразовать комплект в множество. Эта операция возвращает в качестве результата множество `set`, состоящее из элементов комплекта без повторений. Например, если комплект равен `{1, 3, 4, 3, 5, 2, 1}`, то выражение `комплект->asSet` равно значению `{1, 3, 4, 5, 2}` или значению `{1, 2, 3, 4, 5}`. Следует отметить, что порядок записи элементов этого множества не имеет значения, поэтому оба полученных результата являются эквивалентными.
- `последовательность->asSet : set`. Операция `asSet` также позволяет преобразовать последовательность в множество. Эта операция возвращает в качестве результата множество `set`, состоящее из элементов исходной последовательности без учета порядка и без повторений. Например, если последовательность равна `{1, 1, 2, 3, 3, 4, 5}`, то выражение `последовательность->asSet` равно значению `{1, 3, 4, 5, 2}` или значению `{1, 2, 3, 4, 5}`. При этом оба полученных результата являются эквивалентными.

- `последовательность->asBag : bag`. Операция `asBag` позволяет преобразовать последовательность в комплект. Эта операция возвращает в качестве результата комплект `bag`, состоящий из всех элементов исходной последовательности без учета порядка, но с учетом повторений. Например, если последовательность равна `{1, 1, 2, 3, 3, 4, 5}`, то выражение `последовательность->asBag` равно значению `{1, 3, 1, 4, 3, 5, 2}` или значению `{1, 3, 2, 3, 1, 4, 5}`. При этом оба полученных результата также являются эквивалентными.

Аналогично определяются и другие операции преобразования типов `комплект->asSequence`, `множество->asSequence`, `множество->asBag`.

Примеры записи выражений языка OCL

Определение значения переменной

Для определения отдельной переменной используется оператор `let`. В следующем примере определяется переменная `доход` для типа `сотрудник`:

```
context Сотрудник inv:
      let доход : Integer = self.работа.зарплата.
```

В этом случае доход сотрудника полностью определяется его зарплатой.

Определение возраста сотрудника

Для определения возраста сотрудника необходимо выполнить простое арифметическое действие — вычесть год рождения сотрудника из текущего года:

```
context Сотрудник inv:
      let возраст : Integer = текущая_дата.год - self.год_рождения
```

Определение кратности значений

Для определения кратности значений используется операция `size`. Например, чтобы задать ограничение на тип компании, у которой может быть единственный директор, можно использовать следующее выражение языка OCL:

```
context Компания inv:
      self.директор ->size = 1
```

Определение совокупности инвариантов

Наиболее распространенным способом применения языка OCL является спецификация целой совокупности инвариантов для одного и того же класса. В этом случае контекст выражения записывается только один раз, после че-

го следуют записи всех относящихся к этому контексту выражений. Например, следующее выражение определяет особенности некоторого класса `Компания`, который используется для моделирования конкретной предметной области:

```
context Компания
      inv: self.директор ->size = 1
      inv: self.сотрудник ->size >= 50
      inv: self.директор.возраст >= 40
      inv: self.сотрудник >= forAll(зарплата >= 500$)
```

Последнее условие определяет ограничение, согласно которому зарплата любого сотрудника такой компании не может быть менее 500$ (в общем-то и неплохая компания).

В заключение этого краткого обзора основных синтаксических конструкций языка OCL следует привести пример записи ограничений в метамодели самого языка UML. Так, отмеченное выше условие единственности имен параметров и классификаторов в пространстве имен записывается в следующем виде:

```
self.parameter->forAll(p1, p2 | p1.name = p2.name implies p1 = p2)
```

Здесь используется специальная логическая операция `implies`, которая служит обозначением логического следования. Аналогично записываются и другие ограничения, которым должны удовлетворять различные элементы метамодели языка UML.

Глоссарий

Настоящий глоссарий поясняет содержательный смысл основных понятий языка UML, которые используются для спецификации различных представлений модели и построения канонических диаграмм. Некоторые из терминов имеют статус принятого стандарта OMG, другие традиционно применяются для ООАП и ООП. Все термины расположены в алфавитном порядке, в скобках приводится эквивалент каждого из терминов на языке оригинала.

***n*-арная ассоциация** (*n*-ary association) — ассоциация между тремя и большим числом классов. Каждый экземпляр такой ассоциации представляет собой упорядоченный набор (кортеж), содержащий *n* экземпляров из соответствующих классов.

Абстрактный класс (abstract class) — класс, который не имеет экземпляров или объектов.

Абстракция (abstraction) — характеристика сущности, которая отличает ее от других сущностей. Абстракция определяет границу представления соответствующего элемента модели.

Агрегат (aggregate) — класс, который представляет "целое" в отношении "часть — целое".

Агрегация (aggregation) — специальная форма ассоциации, которая служит для представления отношения типа "часть — целое" между агрегатом (целое) и его составной частью.

Актер (actor) — согласованное множество ролей, которые играют внешние сущности по отношению к вариантам использования при взаимодействии с ними. Каждый актер играет единственную роль при взаимодействии с отдельным вариантом использования.

Активация, активизация (activation) — выполнение некоторого действия объектом или подсистемой.

Активный класс (active class) — класс, экземпляры которого являются активными объектами.

Активный объект (active object) — объект, который имеет собственную нить управления и может инициировать деятельность по управлению. Является экземпляром активного класса.

Анализ (analysis) — отдельный этап процесса разработки программной системы, главная цель которого — построить модель предметной области решаемой проблемы. Анализ раскрывает смысл того, что необходимо делать, а проектирование — как это следует делать.

Артефакт (artifact) — семантически законченная часть информации, которая используется или создается в процессе разработки программной системы. Артефакт может быть моделью, графическим элементом, диаграммой или программой.

Архитектура (architecture) — организационная структура и связанное с ней поведение системы. Архитектура допускает рекурсивную декомпозицию на части, взаимодействующие посредством своих интерфейсов, на отношения, связывающие эти части, и на ограничения, соединяющие эти части в систему.

Аспект модели (model aspect) — некоторый взгляд на модель с определенной точки зрения, позволяющий выделить одни характеристики системы и исключить из рассмотрения другие. Например, структурный аспект модели обращает внимание на характеристики структуры и не учитывает другие ее особенности, такие как поведение.

Аспект поведения модели (behavioral model aspect) — аспект модели, который рассматривает поведение отдельных элементов системы, включая методы, кооперации и состояния.

Ассоциация (association) — семантическое отношение между двумя и более классификаторами, которое специфицирует характер связи между соответствующими экземплярами этих классификаторов.

Атрибут (attribute) — содержательная характеристика классификатора, описывающая множество значений, которые могут принимать отдельные экземпляры классификатора.

Базовый тип (primitive type) — предопределенный в языке UML тип данных, не содержащий никакой вложенной структуры. Например, целое число или строка.

Бизнес-актер (business actor) — индивидуум, группа, организация, компания или система, которые являются инициаторами бизнес-вариантов использования для моделируемой системы, но организационно в нее не входят.

Бизнес-вариант использования (business use case) — вариант использования, определяющий последовательность действий моделируемой системы, направленных на выполнение отдельного бизнес-процесса.

Бизнес-сущность (business entity) — пассивный класс, который не инициирует никаких сообщений на диаграммах взаимодействия и информация об экземплярах которого должна сохраняться в моделируемой системе в качестве результата выполнения бизнес-процесса.

Бинарная ассоциация (binary association) — ассоциация между двумя классами. Является специальным случаем *n*-арной ассоциации.

Булев тип (boolean type) — перечисление, имеющее только два значения: истина и ложь.

Булево выражение (boolean expression) — выражение, которое может принимать только одно из Булевых значений.

Вариант использования (use case) — внешняя спецификация последовательности действий, которые система или другая сущность могут выполнять в процессе взаимодействия с актерами.

Вершина (vertex) — в контексте языка UML обозначает источник или цель перехода в конечном автомате. Вершиной может быть состояние или псевдосостояние.

Взаимодействие (interaction) — спецификация способности одних объектов оказывать влияние на поведение других объектов.

Видимость (visibility) — совокупность конкретных значений (public, protected или private), характеризующих способность оказывать влияние на элемент модели со стороны других объектов модели.

Включение (include) — в языке UML отношение между базовым вариантом использования и его специальным случаем. Применяется для спецификации поведения базовых элементов на основе включаемых в него других элементов.

Внутренний переход (internal transition) — переход, специфицирующий реакцию на некоторое событие без изменения состояния объекта.

Временное событие (time event) — событие, которое происходит спустя некоторое время после перехода объекта в соответствующее состояние.

Время выполнения (run time) — период, в течение которого выполняется программа на компьютере.

Входное действие (entry action) — действие, которое выполняется в момент перехода конечного автомата в новое состояние.

Вызов (call) — состояние действия, которое запрашивает выполнение некоторой операции над классификатором.

Выражение (expression) — строка текста, при оценивании которой должно получиться значение определенного типа. Например, выражение " (7 + 5 * 3)" оценивается как целое число.

Выражение времени (time expression) — выражение, которое принимает значение абсолютного или относительного времени.

Выражение типа (type expression) — выражение, результат вычисления или оценки которого является ссылкой на один или несколько типов.

Граничный класс (boundary class) — класс, который располагается на границе системы с внешней средой и непосредственно взаимодействующий с актерами.

Граф деятельности (activity graph) — специальный случай конечного автомата, который используется для моделирования процессов, включающих один или большее число классификаторов.

Действие (action) — спецификация выполнимого утверждения, которая образует абстракцию некоторой вычислительной процедуры. Действие обычно приводит к изменению состояния системы и может быть реализовано посредством передачи сообщения объекту, модификацией существующей связи или значения атрибута.

Действие выхода (exit action) — действие, выполняемое при выходе из состояния конечного автомата.

Делегирование (delegation) — способность одного объекта передавать сообщение другому объекту в ответ на запрос некоторого сервиса.

Диаграмма (diagram) — графическое представление совокупности элементов модели в форме связного графа, вершинам и ребрам (дугам) которого приписывается определенная семантика. Нотация канонических диаграмм является основным средством разработки моделей на языке UML.

Диаграмма вариантов использования (use case diagram) — диаграмма, на которой изображаются отношения между актерами и вариантами использования.

Диаграмма взаимодействия (interaction diagram) — общее название канонических диаграмм кооперации и последовательности. Первоначально использовалась для представления общих аспектов взаимодействия элементов модели системы.

Диаграмма классов (class diagram) — диаграмма, на которой представлена совокупность декларативных (статических) элементов модели, таких как классы, типы и связывающие их отношения.

Диаграмма компонентов (component diagram) — диаграмма, представляющая состав и зависимости между компонентами программной системы.

Диаграмма кооперации (collaboration diagram) — диаграмма, показывающая взаимодействия элементов модели в рамках статической структуры системы. Хотя имеет некоторое концептуальное сходство с диаграммой последовательности, диаграмма кооперации визуализирует отношения между экземплярами.

Диаграмма объектов (object diagram) — диаграмма, на которой представлены объекты и отношения между ними в фиксированный момент. Эта диаграмма может быть частным случаем диаграммы классов или диаграммы кооперации.

Диаграмма последовательности (sequence diagram) — диаграмма, на которой показаны взаимодействия объектов, упорядоченные по времени их проявления. На диаграмме последовательности явно указывается ось времени, позволяющая визуализировать временные отношения между передаваемыми сообщениями.

Диаграмма размещения (deployment diagram) — диаграмма, на которой представлены узлы выполнения программных компонентов реального времени, а также процессов и объектов.

Диаграмма состояний (statechart diagram) — диаграмма, которая представляет некоторый конечный автомат.

Дорожка (swimlane) — отдельная часть разбиения диаграммы деятельности по принципу ответственности отдельных объектов за выполнение соответствующих действий.

Зависимость (dependency) — отношение между двумя элементами модели, при котором изменение одного элемента (независимого) приводит к изменению другого элемента (зависимого).

Значение (value) — отдельный элемент из допустимой области значений переменной заданного типа.

Имя (name) — строка текста, которая используется для идентификации элемента модели.

Инстанцирование (binding) — создание элемента модели на основе шаблона посредством подстановки аргументов вместо параметров шаблона.

Интерфейс (interface) — именованное множество операций, которые характеризуют поведение отдельного элемента модели.

Класс (class) — описание множества однородных объектов, имеющих одинаковые атрибуты, операции, методы, отношения с другими объектами и семантику. Класс может использовать множество интерфейсов для спецификации совокупности операций, которые он предоставляет своему окружению.

Класс ассоциации (association class) — модельный элемент, который одновременно является ассоциацией и классом. Класс ассоциации может рассматриваться как ассоциация, которая обладает свойствами класса, или как класс, имеющий также свойства ассоциации.

Класс-сущность (entity class) — пассивный класс, информация о котором должна храниться постоянно и не уничтожаться с выключением системы.

Классификатор (classifier) — механизм описания свойств поведения или структуры. Классификаторами являются интерфейсы, классы, типы данных и компоненты.

Классификация (classification) — установление соответствия между некоторым объектом и классификатором.

Клиент (client) — классификатор, который требует некоторого сервиса от другого классификатора.

Композит (composite) — класс, который связан отношением композиции с одним или бóльшим числом классов.

Композиция (composition) — форма ассоциации агрегации, при которой составные части целого имеют такое же время жизни, что и само целое. Эти части уничтожаются вместе с уничтожением целого.

Компонент (component) — физически существующая часть системы, которая обеспечивает исполнение пакетов и предусматривает реализацию множества интерфейсов. Компонентом может быть исполняемый код отдельного модуля, командные файлы или файлы, содержащие интерпретируемые скрипты.

Конец ассоциации (association end) — конечная точка ассоциации, которая связывает ассоциацию с классификатором.

Конец связи (link end) — отдельный экземпляр конца ассоциации.

Конечное состояние (final state) — разновидность псевдосостояния, обозначающая прекращение процесса изменения состояний конечного автомата или нахождения моделируемого объекта в составном состоянии.

Конечный автомат (state machine) — модель для спецификации поведения объекта в форме последовательности его состояний, которые описывают реакцию объекта на внешние события, выполнение объектом некоторых действий, а также изменение его отдельных свойств.

Конкретный класс (concrete class) — класс, на основе которого могут быть непосредственно созданы экземпляры (объекты).

Контейнер (container) — экземпляр или компонент, который включает в качестве своих частей другие экземпляры или компоненты.

Контекст (context) — представление множества относящихся к модели элементов, выполненное с определенной целью.

Кооперация (collaboration) — спецификация множества ролей и других элементов, совместно взаимодействующих с целью реализации отдельных операций или деятельностей.

Кратность (multiplicity) — спецификация области значений допустимой мощности, которой могут обладать соответствующие множества. В языке UML кратность широко используется для задания ролей ассоциаций, составных объектов и значений атрибутов.

Линия жизни объекта (object lifeline) — вертикальная линия на диаграмме последовательности, которая представляет существование объекта в течение определенного периода.

Метакласс (metaclass) — класс, экземпляры которого также являются классами. Метаклассы используются обычно для построения метамоделей.

Мета-метамодель (meta-metamodel) — модель, предназначенная для определения языка описания метамодели. Отношение между мета-метамоделью и метамоделью аналогично отношению между метамоделью и моделью.

Метамодель (metamodel) — модель, предназначенная для определения языка описания модели.

Метаобъект (metaobject) — общее название всех метасущностей в метамодели языка UML. Примерами метаобъектов являются мататипы, метаклассы, метаатрибуты и метаассоциации.

Метод (method) — конкретная реализация операции. Как правило, метод специфицирует конкретный алгоритм или процедуру, ассоциированную с соответствующей операцией.

Модель (model) — абстракция физической системы, рассматриваемая с определенной точки зрения. Одна и та же физическая система может быть представлена несколькими моделями, что отражает общий принцип многомодельности системного анализа. Как правило, назначение отдельной модели системы определяется характером решаемой проблемы.

Модель вариантов использования (use case model) — модель, которая описывает функциональные требования к системе в терминах вариантов использования.

Модуль (module) — часть программной системы, требующая памяти для своего хранения и процессора для исполнения.

Мощность множества (cardinality) — число элементов конечного множества. Для бесконечных множеств имеет более сложную семантику.

Наследование (inheritance) — специальный концептуальный механизм, посредством которого более специальные элементы включают в себя структуру и поведение более общих элементов.

Наследование интерфейса (interface inheritance) — наследование интерфейса более общего элемента. Не включает в себя наследование реализации этого элемента.

Начальное состояние (final state) — разновидность псевдосостояния, обозначающее начало выполнения процесса изменения состояний конечного автомата или нахождения моделируемого объекта в составном состоянии.

Несовместимое подсостояние (disjoint substate) — подсостояние, в котором подсистема не может находиться одновременно с другими подсостояниями одного и того же составного состояния.

Нить <управления> (thread <of control>) — некоторый простой путь выполнения программы, динамической модели или любого другого потока управления. Служит для обозначения некоторого процесса, не требующего отдельных системных ресурсов.

Обобщение (generalization) — таксономическое отношение между более общим понятием и менее общим понятием. Менее общее понятие должно быть согласовано с более общим и содержит дополнительную информацию.

Объект (object) — сущность с хорошо определенными границами и индивидуальностью, которая инкапсулирует состояние и поведение. В контексте языка UML является экземпляром некоторого класса.

Ограничение (constraint) — некоторое логическое условие, ограничивающее семантику выбранного элемента модели. Некоторые ограничения предопределены в языке UML, другие могут быть специфицированы самим разработчиком.

Операция (operation) — сервис, который может быть востребован одним объектом у другого. Операция имеет сигнатуру, которая может ограничивать возможные значения действительных ее параметров.

Организационная единица (organization unit) — пакет, в состав которого могут входить сотрудники, бизнес-сущности, бизнес варианты использования, отдельные диаграммы языка UML и другие организационные единицы. Синоним — организационное подразделение.

Отношение (relationship) — семантическая связь между отдельными элементами модели. Примерами отношений в языке UML являются ассоциации и обобщения.

Отправитель, объект-отправитель (sender <object>) — объект, который отправляет некоторую информацию объекту-получателю.

Пакет (package) — некоторый общецелевой механизм для организации артефактов в множество, реализующий системный принцип декомпозиции модели сложной системы и допускающий вложенность пакетов друг в друга.

Параллельное подсостояние (concurrent substate) — подсостояние, в котором подсистема может находиться одновременно с другими параллельными подсостояниями, входящими в одно составное состояние.

Параллельность (concurrency) — возможность одновременного выполнения двух и более видов деятельности в течение некоторого интервала времени.

Параметр (parameter) — спецификация переменной, которая может быть изменена, передана или возвращена. Параметр может включать имя, тип и направление. Используется для задания операций, сообщений и событий.

Параметризуемый элемент, шаблон (parameterized element, template) — описатель класса, который содержит один или несколько неопределенных параметров.

Паттерн (pattern) — параметризованная кооперация вместе с описанием базовых принципов ее использования в форме параметров, которые ограничивают использование объектов в контексте ролей элементов рассматриваемого паттерна.

Передача <сообщения> (send <a message>) — прохождение информации от отправителя к получателю.

Переход (transition) — отношение между двумя состояниями, которое указывает, что объект в первом состоянии должен выполнить определенные действия и перейти во второе состояние. При этом должно произойти заданное событие и должны быть выполнены некоторые дополнительные условия. Когда изменяется состояние системы, то говорят о срабатывании перехода.

Перечисление (enumeration) — список именованных значений, используемых в качестве области значений атрибута отдельного типа. Например, `Цвет = {красный, зеленый, голубой}`.

Поведение (behavior) — наблюдаемый результат выполнения операции или наступления некоторого события.

Подкласс (subclass) — специализация некоторого класса в отношении обобщения.

Подпакет (subpackage) — пакет, который является составной частью другого пакета.

Подсистема (subsystem) — совокупность модельных элементов, которая представляет фрагмент поведения физической системы. Подсистема представляется своими интерфейсами и имеет операции. Модельные элементы подсистемы могут быть разделены на элементы спецификации и элементы реализации.

Подсостояние (substate) — состояние, которое является частью составного состояния. Например, параллельное состояние, исключающее состояние.

Подтип (subtype) — классификатор, который является специализацией более общего классификатора типа в отношении обобщения.

Получатель <объект-получатель> (receiver <object>) — объект, который наделен способностью воспринимать информацию, передаваемую другими объектами.

Получать <сообщение> (receive <a message>) — способность одного элемента модели воспринимать информацию от других элементов.

Помеченное значение (tagged value) — явное определение свойства как пары "имя — значение". В помеченном значении само имя называют *тегом* (tag). Отдельные теги предопределены в языке UML, другие могут быть определены пользователем. Помеченные значения являются одним из трех механизмов расширения языка UML.

Поставщик, сервер (supplier) — классификатор, который предоставляет некоторые сервисы другим классификаторам-клиентам.

Постусловие (postcondition) — ограничение, которое должно быть истинным для завершения операции.

Потомок (child) — специализация одного из элементов отношения обобщения, называемого в этом случае родителем. Примерами потомков являются подкласс, подсистема.

Предметная область (domain) — область знаний или некоторой деятельности, характеризуемая множеством понятий и терминологией, понятной специалистам в этой предметной области.

Представление (view) — проекция модели, которая рассматривается с определенной точки зрения. В отдельном представлении учитываются только существенные аспекты модели, а все второстепенные детали могут быть опущены.

Предусловие (precondition) — ограничение, которое должно быть истинным для начала выполнения операции.

Примечание (comment) — элемент графической нотации, присоединенный к другому элементу модели и не имеющий собственной семантики.

Проектирование (design) — часть процесса разработки программной системы, основная цель которой — решить, как система должна быть реализована, чтобы удовлетворять предъявляемым к ней требованиям.

Проекция (projection) — отображение некоторого множества в некоторое его подмножество.

Проекция представления (view projection) — некоторая проекция элементов модели на элементы представления. Проекция представления обеспечивает расположение и стиль каждому элементу представления.

Производный элемент (derived element) — элемент модели, который может быть вычислен посредством другого элемента, но тем не менее включается в модель для ясности или удобства разработки, хотя при этом в модель не вносится никакой новой семантики.

Простое наследование (single inheritance) — семантический вариант обобщения, при котором тип может иметь только один супертип.

Пространство имен (namespace) — часть модели, в которой определяются и используются имена элементов модели. В одном пространстве имен каждое имя должно быть уникальным (единственным).

Процесс (process) — самостоятельная единица параллельности и исполнения в операционной системе. Применительно к ООАП процесс разработки программной системы представляет собой последовательность взаимосвязанных этапов, направленных на реализацию проекта системы.

Процесс разработки (development process) — множество частично упорядоченных шагов, каждый из которых направлен на решение отдельной задачи в ходе процесса конструирования и реализации моделей.

Псевдосостояние (pseudo-state) — вершина в конечном автомате, которая имеет форму состояния, но не обладает поведением. Псевдосостояниями являются начальное и конечное состояния.

Разбиение (partition) — в графе деятельности область, содержащая группу отдельных под-деятельностей с общей ответственностью за их выполнение. Применительно к архитектуре модели множество соответствующих классификаторов или пакетов одного уровня абстракции или одной подсистемы на разных уровнях.

Родитель, предок (parent) — в отношении обобщения более общий элемент.

Роль (role) — имеющее имя специфическое поведение некоторой сущности, рассматриваемой в определенном контексте. Роль может быть статической или динамической.

Свойство (feature) — характеристика системы, такая как операция или атрибут, которая служит для спецификации классификаторов в языке UML.

Свойство (property) — именованное значение, которое обозначает характеристику некоторого элемента. Свойство имеет собственную семантику. Некоторые свойства предопределены в языке UML, другие могут быть определены разработчиком.

Свойство поведения (behavioral feature) — динамическая характеристика элемента модели, такая как операция или метод.

Связь (link) — любое семантическое отношение между некоторой совокупностью объектов. Является экземпляром ассоциации.

Сигнал (signal) — спецификация асинхронных сообщений между объектами. Сигналы могут иметь параметры.

Сигнатура (signature) — имя и параметры некоторого свойства поведения. Сигнатура может включать необязательный возвращаемый параметр.

Система (system) — в метамодели языка UML подсистема самого верхнего уровня абстракции.

Событие (event) — спецификация существенных явлений в поведении системы, которые имеют местоположение во времени и пространстве.

Сообщение (message) — спецификация передачи информации от одного элемента модели к другому с ожиданием выполнения определенных действий со стороны принимающего элемента.

Составное состояние (composite state) — состояние, имеющее в качестве внутренних подсостояний параллельные или последовательные состояния.

Состояние (state) — условие или ситуация в ходе жизненного цикла объекта, в течение которого он удовлетворяет некоторому условию, выполняет некоторую деятельность или ожидает некоторого события.

Состояние действия (action state) — состояние, которое представляет выполнение некоторого атомарного действия, такого как обращение к операции.

Состояние подавтомата (submachine state) — состояние в конечном автомате, которое эквивалентно составному состоянию, отдельные элементы которого описываются другим конечным автоматом.

Состояние под-деятельности (subactivity state) — состояние в графе деятельности, которое служит для представления неатомарной последовательности шагов в некотором направлении.

Состояние потока объекта (object flow state) — состояние в графе деятельности, которое представляет переход объекта от выполнения действий выхода из одного состояния к выполнению входных действий в другом состоянии.

Состояние синхронизации (synch state) — вершина в конечном автомате, которая используется для синхронизации параллельных областей конечного автомата.

Сотрудник (business worker) — индивидуум, который действует внутри моделируемой системы (организации), взаимодействует с другими сотрудниками и манипулирует бизнес-сущностями в процессе выполнения отдельных операций бизнес-процесса.

Спецификация (specification) — декларативное описание некоторой сущности или поведения.

Срабатывание <перехода> (fire) — выполнение перехода из одного состояния в другое состояние.

Статическая классификация (static classification) — семантический вариант обобщения, при котором объект не может изменять классификатор.

Стереотип (stereotype) — новый тип элемента модели, который расширяет семантику метамодели. Стереотипы должны основываться на уже существующих и описанных в метамодели языка UML типах или классах. Стереотипы предназначены для расширения именно семантики, но не структуры уже описанных типов или классов. Некоторые стереотипы предопределены в языке UML, другие могут быть определены пользователем. Являются одним из трех механизмов расширения в языке UML.

Сторожевое условие (guard condition) — условие, которое должно быть обязательно выполнено для того, чтобы сработал ассоциированный с ним переход.

Строка (string) — последовательность символов текста. Набор конкретных допустимых символов зависит от реализации и, как правило, включает в себя интернациональную кодировку алфавита и символов графики.

Структурное свойство (structural feature) — статическое свойство элемента модели, такое как атрибут.

Структурный аспект модели (structural model aspect) — аспект модели, который обращает внимание только на структуру объектов в системе, включая их типы, классы, отношения, атрибуты и операции.

Суперкласс (superclass) — более общий класс в отношении обобщения.

Супертип (supertype) — более общий классификатор типа в отношении обобщения.

Сценарий (scenario) — определенная последовательность действий, которая описывает поведение моделируемой системы в форме обычного текста. В контексте языка UML сценарий используется для дополнительной иллюстрации взаимодействия актеров и вариантов использования.

Тип (type) — стереотип класса, который используется для спецификации области значений экземпляров (объектов) совместно с операциями, применимыми к этим объектам.

Тип данных (datatype) — описатель множества значений, которые могут принимать атрибуты классов и возвращать в качестве значений операции классов.

Требование (requirement) — желательное свойство, характеристика или поведение системы.

Узел (node) — классификатор, который представляет собой некоторый вычислительный ресурс времени исполнения программы. Узел должен обладать по меньшей мере памятью, а возможно, содержать процессор.

Управляющий класс (control class) — класс, отвечающий за координацию действий других классов. На каждой диаграмме классов должен быть хотя бы один управляющий класс, причем объектам управляющего класса посылают мало сообщений, а он рассылает много сообщений.

Уровень (layer) — организация классификаторов или пакетов на одном уровне абстракции. Уровень представляет горизонтальный срез архитектуры модели, в то время как разбиение представляет ее вертикальный срез.

Физическая система (physical system) — реально существующий прототип модели системы. В контексте языка UML обозначает совокупность связанных физических сущностей, включая программное и аппаратное обеспечение, а также персонал, которые организованы для выполнения специальных задач.

Фокус управления (focus of control) — специальный символ на диаграмме последовательности, указывающий период, в течение которого объект выполняет некоторое действие.

Экземпляр (instance) — сущность, которая является конкретным представителем классификатора.

Экземпляр варианта использования (use case instance) — исполнение последовательности действий, специфицированных в некотором варианте использования.

Элемент (element) — атомарная составляющая модельного представления.

Элемент модели (model element) — элемент, который является некоторой абстракцией, извлеченной из моделируемой системы.

Элемент представления (view element) — текстовая и/или графическая проекция некоторой совокупности элементов модели.

Этап анализа (analysis time) — часть процесса ООАП, относящаяся к выполнению фазы анализа процесса разработки программной системы.

В заключение приводится список встречающихся в книге общепринятых сокращений с кратким пояснением их содержательного смысла.

ATL (Microsoft Active Template Library) — библиотека активных шаблонов, которая была специально создана с целью обеспечения большей простоты и гибкости при разработке объектов COM на языке программирования C++.

CASE (Computer Aided Software Engineering) — автоматизированная разработка программного обеспечения. Методология разработки программ, основанная на комплексном использовании компьютеров не только для написания исходных кодов, но и для анализа и моделирования соответствующей предметной области.

CBD (Component-Based Development) — компонентная разработка приложений. Новая парадигма разработки программ, интенсивно развиваемая в последнее время. Основные идеи этой парадигмы во многом созвучны с технологией CORBA.

CORBA (Common Object Request Broker Architecture) — общая архитектура брокеров объектных запросов. Этим термином обозначают технологию, архитектуру и набор стандартов для создания распределенных программных приложений. Является главным стратегическим направлением деятельности консорциума OMG. Оценивается специалистами как новый многообещающий подход к построению программных систем, который предполагает использование готовых компонентов и предлагает технологию их интеграции.

DFD (Data Flow Diagram) — диаграмма потоков данных. Графическая нотация, ориентированная на создание моделей распределенной обработки данных.

ERD (Entity-Relation Diagram) — диаграмма "сущность-связь". Графическая нотация, предложенная П. Ченом и получившая дальнейшее развитие в работах Р. Баркера.

IDL (Interface Definition Language) — язык определения интерфейса. Специальный декларативный язык для описания стандартов CORBA, для которого существуют компиляторы в большинство современных языков программирования.

MTS (Microsoft Transaction Server) — сервер транзакций Microsoft, представляющий собой основанную на компонентах систему обработки транзакций, предназначенную для разработки масштабируемых сетевых приложений.

OCL (Object Constraint Language) — язык объектных ограничений. Составная часть унифицированного языка моделирования UML. Служит для записи ограничений и правил правильного построения выражений UML.

OMG (Object Management Group) — название консорциума, созданного в 1989 году для разработки индустриальных стандартов с их последующим

использованием в процессе создания интероперабельных неоднородных распределенных объектных сред.

OMT (Object Modeling Technique) — методология и графическая нотация объектно-ориентированного анализа и проектирования, разработчиком которых является Джеймс Румбах (James Rumbaugh) и которая реализована в CASE-средстве Rational Rose 98.

OOA/OOD, OOA&D (Object-Oriented Analysis/Design) — объектно-ориентированный анализ и проектирование (ООАП). Этим термином обозначают технологию разработки программных систем, в основу которых положена парадигма представления окружающего нас мира в виде объектов, являющихся экземплярами соответствующих классов.

OOP (Object-Oriented Programming) — объектно-ориентированное программирование (ООП). Совокупность принципов, технологии и инструментальных средств для создания программных систем, в основу которых закладывается архитектура взаимодействия объектов. Говоря об ООП, чаще всего отмечают его основные принципы: абстракция, наследование, полиморфизм и инкапсуляция.

OOSE (Object-Oriented Software Engineering) — методология и элементы графической нотации объектно-ориентированного анализа и проектирования, одним из основных разработчиков которых является Айвар Джекобсон (Ivar Jacobson).

RUP (Rational Unified Process) — рациональный унифицированный процесс, совокупность понятий и принципов, позволяющих организовать процесс ООАП на основе некоторой методики, которая использует отдельные элементы нотации языка UML.

SADT (Structured Analysis and Design Technique) — технология структурного анализа и проектирования, разработчиком которой является Дуглас Росс (D. Ross).

UML (Unified Modeling Language) — унифицированный язык моделирования для описания, визуализации и документирования объектно-ориентированных систем в процессе их разработки.

Литература

1. Амриш К. И., Ахмед Х. З. Разработка корпоративных Java-приложений с использованием J2EE и UML. Пер. с англ. — М.: "Вильямс", 2002.— 272 с.
2. Бадд Т. Объектно-ориентированное программирование в действии: Пер. с англ. — СПб.: "Питер", 1997. — 464 с.
3. Бен-Ари М. Языки программирования. Практический сравнительный анализ: Пер. с англ. — М.: Мир, 2000. — 366 с.
4. Боггс У., Боггс М. UML и Rational Rose: Пер. с англ. — М.: "ЛОРИ", 2000. — 582 с.
5. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++: Пер. с англ. — М.: "Бином"; СПб.: "Невский диалект", 1999. — 560 с.
6. Буч Г., Рамбо Дж., Джекобсон А. Язык UML. Руководство пользователя: Пер. с англ. — М.: ДМК, 2000. — 432 с.
7. Вендров А. М. CASE-технологии. Современные методы и средства проектирования информационных систем. — М.: "Финансы и статистика", 1998. — 176 с.
8. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования: Пер. с англ. — СПб.: "Питер", 2001.— 368 с.
9. Гома Х. UML. Проектирование систем реального времени, параллельных и распределенных приложений. Пер. с англ. — М.: "ДМК Пресс", 2002. — 704 с.
10. Грегори К. Использование Visual C++ 6. Специальное издание: Пер. с англ. — М., СПб., Киев: "Вильямс", 1999. — 864 с.

11. Йордон Э., Аргила К. Структурные модели в объектно-ориентированном анализе и проектировании: Пер. с англ. — М.: "Лори", 1999. — 264 с.

12. Калянов Г. Н. CASE-структурный системный анализ (автоматизация и применение). — М.: "ЛОРИ", 1996. — 242 с.

13. Кенту М. Delphi 6 для профессионалов: Пер. с англ. — СПб.: "Питер", 2002. — 1088 с.

14. Коналлен Дж. Разработка Web-приложений с использованием UML. Пер. с англ. — М.: "Вильямс", 2001. — 288 с.

15. Коуд П., Норт Д., Мейфилд М. Объектные модели. Стратегии, шаблоны и приложения: Пер. с англ. — М.: "Лори", 1999. — 434 с.

16. Ларман К. Применение UML и шаблонов проектирования. Пер. с англ. — М.: "Вильямс", 2001. — 496 с.

17. Ларман К. Применение UML и шаблонов проектирования. 2-е издание. Пер. с англ. — М.: "Вильямс", 2002. — 624 с.

18. Леоненков А. В. Самоучитель UML. — СПб.: "БХВ — Петербург", 2001. — 304 с.

19. Леффингуэлл Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход. Пер. с англ. — М.: "Вильямс", 2002. — 448 с.

20. Рамбо Дж., Якобсон А., Буч Г. UML: специальный справочник. Пер. с англ. — СПб.: "Питер", 2001. — 656 с.

21. Резников Б. А. Системный анализ и методы системотехники. Часть I. Методология системных исследований. Моделирование сложных систем. — МО СССР, 1990. — 522 с.

22. Розенберг Д., Скотт К. Применение объектного моделирования с использованием UML и анализ прецедентов. Пер. с англ. — М.: "ДМК Пресс", 2002. — 160 с.

23. Свами М., Тхуласираман К. Графы, сети и алгоритмы. — М.: Мир, 1984. — 455 с.

24. Уайт Б. А. Управление конфигурацией программных средств. Практическое руководство по Rational ClearCase. Пер. с англ. — М.: "ДМК Пресс", 2002. — 272 с.

25. Фаулер М., Скотт К. UML. Основы. Пер. с англ. — СПб.: "Символ-Плюс", 2002. — 192 с.

26. Шмуллер Д. Освой самостоятельно UML за 24 часа. Пер. с англ. — М.: "Вильямс", 2002. — 352 с.

27. Kruchten P. The Rational Unified Process. An Introduction. Addison-Wesley, 2000. — 300 p.

28. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. Пер. с англ. — СПб.: "Питер", 2002. — 496 с.

29. Barker R. CASE*Method. Entity-Relationship Modeling. Copyright Oracle Corporation UK Limited, Addison-Wesley Publishing Co., 1990.

30. Chen P. The Entity-Relationship Model — Toward a Unified View of Data. — ACM Transactions on Database Systems, v. 1, no.1, 1976.

31. Cockburn A. Writing Effective Use Cases. Addison-Wesley, 2000.

32. Jacobson I., Booch G., Rumbaugh J. Unified Software Development Process. Addison-Wesley, 1999. — 512 p.

33. Kruchten P. The Rational Unified Process. An Intriduction. Addison-Wesley, 2000. — 300 p.

34. Rumbaugh J., Jacobson I., Booch G. The Unified Modeling Language. Reference Manual. Addison-Wesley, 1999. — 550 p.

Предметный указатель

"

"Черный ящик" 36

<

<<bind>> 158
<<constraint>> 111
<<derive>> 158
<<document>> стереотип 265
<<executable>> стереотип 264
<<extend>> 114
<<extend>> стереотип 324
<<file>> стереотип 264
<<import>> 158
<<include>> 113
<<include>> стереотип 322
<<interface>> 160
<<library>> стереотип 265
<<modem>> стереотип 278
<<net>> стереотип 278
<<printer>> стереотип 278
<<processor>> стереотип 278
<<refine>> 158
<<sensor>> стереотип 278
<<signal>> 144
<<source>> стереотип 265
<<table>> стереотип 265
<<topLevelPackage>> 123
<<useCaseModel>> 123
<<актер>> 109
<<приемопередатчик>> стереотип 282

A

Activity 56
Actor 108
Arrows 56
Association Class 149
Association End 149

B

Booch 63

C

CASE 33
CRC-карточка 31
Class 134

D

DFD (Data Flow Diagram) 59

I

ICAM 55
ICOM 56
IDEF 55

N

N-арная ассоциация 147

O

Object Constraint Language OCL 92
Object Modeling Technique OMT 63
Object-Oriented Software Engineering OOSE 63
OMG 65

R

Rational Unified Process RUP 101
Request For Proposals RTP 65

S

SADT 55

U

UML 34
Use case 106

V

Visibility 137

А

Абстрактная операция 144
Абстрактный класс 136
Абстрактный синтаксис 92
Автоматизированная разработка программного обеспечения 33
Актер 88
Актер (actor) 108
Активный объект 178
Алгоритм 18
Алгоритмическая организация 17
Анонимный объект 176
Артефакты 264
Асинхронное поведение модели 211
Ассоциация-класс (Association Class) 149
Атрибут класса 136

Б

Бизнес-актер 118
Бизнес-вариант использования (business use case) 119
Бизнес-сущность 160
Бинарная ассоциация 146
Блок-схема алгоритма 18
Браузер проекта 295

В

Вариант использования 88
Вариант использования (use case) 106
Верхняя граница 138
Вершина графа 44
Ветвление потока управления 199
Вид параметра 142
Виды компонентов, диаграмма компонентов 263
Внешняя сущность 59
Внутренние переходы 217
Временные ограничения 202
Второстепенный актер 113
Входные воздействия 36
Выражение действия 224 242
Выходные воздействия 36

Г

Генерация программного кода 366, 370
Главное меню Rational Rose 2002 299
Главный актер 113
Глубокое историческое состояние 229
Границы множества 41
Граничный класс 159
Граф 44
Графическая нотация 50
Графические примитивы 97

Д

Действия 87
Дерево граф 47
Деятельность 241
(activity) 56
Диаграмма 95
вариантов использования (use case diagram) 105
Венна 39
взаимодействия 169
деятельности 240
классов (class diagram) 133
компонентов 261
кооперации 169
кооперации уровня примеров 174
развертывания 275
реализации 261
сущность-связь 52
функционального моделирования 55
Дисциплина 121
Добавление:
актера на диаграмму варианта использования 318
ассоциации 321
атрибутов классов 328
варианта использования 320
деятельности на диаграмму деятельности 354
и удаление компонентов на диаграмму компонентов 359
класса 327
объект на диаграмму кооперации 334
объекта на диаграмму последовательности 342
операций классов 330
отношений на диаграмму класов 331
отношения зависимости 322
отношения зависимости на диаграмму компонентов 360
перехода 350
перехода на диаграмму деятельности 355
связи 336
соединения на диаграмму развертывания 365
сообщения 337
сообщения на диаграмму последовательности 343
состояния на диаграмму состояний 348
узла на диаграмму развертывания 363
Дорожки 249
Достижимость состояний 213

Ж

Жизненный цикл программы 32

З

Зависимости, диаграмма компонентов 266
Запись ООП 25

И

Иерархическая структура
модельных представлений 77
Импортируемый интерфейс 266
Имя:
ассоциации 146
атрибута 138
класса 135
компонента, диаграмма
компонентов 262
операции 142
роли 149
состояния 216
Инкапсуляция ООП 27
Инструментарий быстрой
разработки приложений 33
Интерфейс 160
IBM Rational Rose 2002 293
ООП 28
диаграмма компонентов 265
Исключающая ассоциация (Xor-
association) 147
Историческое состояние 229
Исходное значение атрибута 140

К

Квантор видимости (visibility) 137
Класс (class) 134
потомок 24
предок 24
ООП 22
Классификатор 133
Класс-сущность 159
Компонент диаграмма
компонентов 262
Компонента 31
Конец ассоциации (Association
End) 149
Конечное (финальное)
состояние 219
подавтоматы 213
автомат 212
Конфликт триггерных
переходов 222
Концептуализация предметной
области 31
Концептуальная схема (модель) 31
Кооперация 170
Корневая вершина (корень) 47
Кортеж 44
Кратность:
ассоциации 149
атрибута 138

Л

Линия жизни объекта 195
Листья дерева 48

М

Маршрут граф 46
Мастер типовых проектов 316
Метаатрибут 77
Метакласс 77
Мета-метамодель 77
Метамодель 77
Метаоперация 77
Метапакет 82
Метапредставление 71
Метасистема 34
Методология структурного
программирования 20
Механизмы расширения 158
Множество 39
Модели поведения 76
Моделирование 36
Модель 90
в системном анализе 36
вариантов использования 123
Мощность множества 42
Мультиобъект 177

Н

Накопитель данных 61
Направленная бинарная
ассоциация 146

Наследование ООП 24
Настройка шрифтов цвета линий и графических элементов 316
Начальное состояние 218
Неглубокое историческое состояние 229
Ненаправленная бинарная ассоциация 146
Неориентированный граф 45
Нетриггерные переходы 221
Нижняя граница 138
Нисходящее проектирование 20

О

Объединение множеств 42
Объект 176
  ООП 22
Объектно-ориентированная декомпозиция 30
Объектно-ориентированное программирование (ООП) 22
Объекты 193
Объем понятия 23
Окно:
  диаграммы 297
  документации 298
  журнала 298
ООАП 30
Операция (operation) 141
Ориентированный граф 45
Отношение 112
  агрегации 154
  ассоциации 112, 146
  включения 113
  вложенности пакетов 81
  зависимости 157
  композиции 155
  множеств (связь-соотношение) 43
  обобщения 116, 150
  расширения 114
  зависимости диаграмма развертывания 280
  между классами 145
  множеств 42
Отображение классов на компоненты 368

П

Пакет 79
  Варианты использования 88
  Графы деятельности 89
  Действия 89
  Конечные автоматы 88
  Кооперации 87
  Механизмы расширения 84
  Общее поведение 87
  Основные элементы 83
  Система 123
  Типы данных 85
  Управление моделями 90
  Элементы поведения 86
  Элементы ядра 83
Параллельные подсостояния 227
Параллельный переход 230
Параметризованный класс 162
Пассивный объект 178
Пересечение множеств 42
Переход 212
  системы 35
Перечисление 85
Подмножество 41
Подпакет 79
Подсервис 128
Подсистема 34 91
Подсостояния 225
Полиморфизм ООП 29
Последовательные подсостояния 225
Потомок 116
Предложение-условие 185
Предметная область 31
Предок 116
Примечание (note) 111
Принадлежность к множеству 41

Принцип:
абстрагирования 70
иерархического построения моделей 71
многомодельности 70
Проверка модели 367
Простой переход 219
Процедура 18
Процедурная:
декомпозиция 18
организация 17
Процесс 60
Пункт:
меню Add-Ins (Расширения) 311
меню Browse (Обзор) 305
меню Edit (Редактирование) 301
меню File (Файл) 299
меню Format (Формат) 304
меню Help (Справка) 311
меню Tools (Инструменты) 308
меню View (Вид) 302
меню Window (Окно) 311
Путь граф 47

Р

Разделение 231, 247
Рациональный унифицированный процесс 101
Реактивные системы 211
Реализация ООП 28
Ребро графа 44
Рекуррентность 185
Ресурсоемкий узел 279
Рефлексивная ассоциация 146
Решение (decision) 243
Роли стрелок 56
Роль участника 171

С

Свойства генерации программного кода 370
Связь (relationship) 52, 180
Секция:
атрибутов 136
операций 141
Семантические сети 49
Сигналы 87
Синхронизирующее состояние 232
Сирота 176
Система (понятие) 34
Системный структурный анализ ССА 51
Слияние 247
Сложность системы 37
Событие 220
Содержание понятия 23
Соединения, диаграмма развертывания 280
Сообщение 182
Соотношение множеств 43
Составное состояние 225
Составной объект 179
Состояние 212, 215
действия 241
под-деятельности 242
системы 35
композит 225
Сотрудник 118, 159
для связи с окружением 159
Специальная панель инструментов 296
Список:
параметров 142
свойств 85
Стандартная панель инструментов 294
Стереотипы 80
связей 181
сообщений 187
Сторожевое условие 221
Стрелка-обобщение 116
Стрелки (arrows) 56
Строка-свойство атрибута 141
Структура:
системы 34
ООП 25

Структурные модели 76
СУБД 33
Суперсервис 128
Суперсостояние 225
Сущность 52
Сценарий 107
Счетная мощность 43

Т

Текст-сценарий 107
Теория множеств 39
Терминатор 60
Тернарная ассоциация 147
Тип атрибута 139
Точки динамического выбора 238
Точки соединения 238
Триггерные переходы 221

У

Узел, диаграмма развертывания 276
Управляющий класс 158
Устройство, диаграмма развертывания 279

Ф

Фокус управления 196
Формат записи сообщений 183
Формирование проектной документации 365

Ш

Шаблон 162

Э

Экземпляр сущности 52
Экспортируемый интерфейс 266
Элементы множества 40

Я

Язык объектных ограничений 92